Chapter 11
函數設計

11-1 Python函數基本觀念

- 內建函數
  - print(), id(), type(), max(), min(), ..
- 程式中不斷重複出現或是一段完整功能的程式片段
  - 整理成函數，方便後續直接呼叫或是方便程式設計師理解。
  - 模組化程式
- 如何設計自己的函數？

11-1 Python函數基本觀念

- 函數的定義：

```
1    def 函數名稱(參數值1[, 參數值2, …]):
2        '''函數註解區塊(docstring)'''
3        程式碼區塊                          # 需要內縮
4        return 回傳值1, 回傳值2, ...        # 回傳值，可以有多個
```

  - 函數名稱：
    - 必須是唯一的，可被呼叫引用。命名規則與一般變數相同。不過 PEP 8 建議英文字母使用小寫。
  - 參數值：
    - 可有可無，參數間用 “,” 隔開。
  - 函數註解：
    - 可有可無，但大型程式中最好都要有，方便閱讀。
  - return：
    - 可有可無，參數間用 “,” 隔開。

11-1 Python函數基本觀念

- 沒有傳入參數也沒有傳回值的函數

```
print("Python 歡迎你")
print("祝大家學習順利")
print("成為 Python 高手")
print("Python 歡迎你")
print("祝大家學習順利")
print("成為 Python 高手")
print("Python 歡迎你")
print("祝大家學習順利")
print("成為 Python 高手")
print("Python 歡迎你")
print("祝大家學習順利")
print("成為 Python 高手")
print("Python 歡迎你")
print("祝大家學習順利")
print("成為 Python 高手")
```

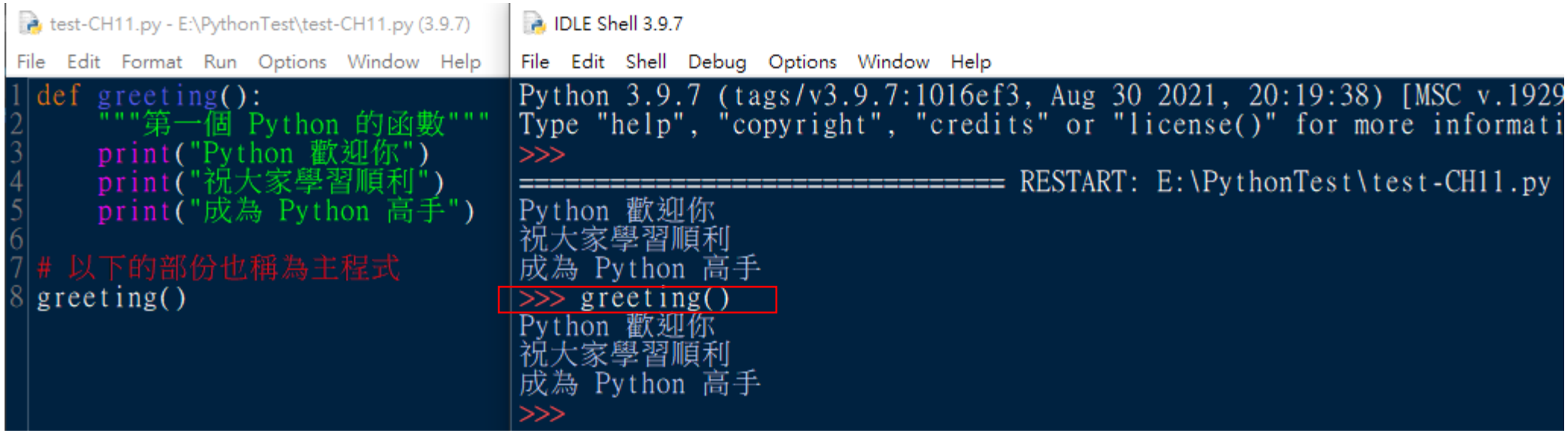
```
Python 歡迎你
祝大家學習順利
成為 Python 高手
Python 歡迎你
祝大家學習順利
成為 Python 高手
Python 歡迎你
祝大家學習順利
成為 Python 高手
Python 歡迎你
祝大家學習順利
成為 Python 高手
Python 歡迎你
祝大家學習順利
成為 Python 高手
```

```
def greeting():
    """第一個 Python 的函數"""
    print("Python 歡迎你")
    print("祝大家學習順利")
    print("成為 Python 高手")

# 以下的部份也稱為主程式
greeting()
greeting()
greeting()
greeting()
greeting()
```

11-1 Python函數基本觀念

- 執行完程式後，在 IDLE Shell 裡面輸入 greeting()，也會印出結果。

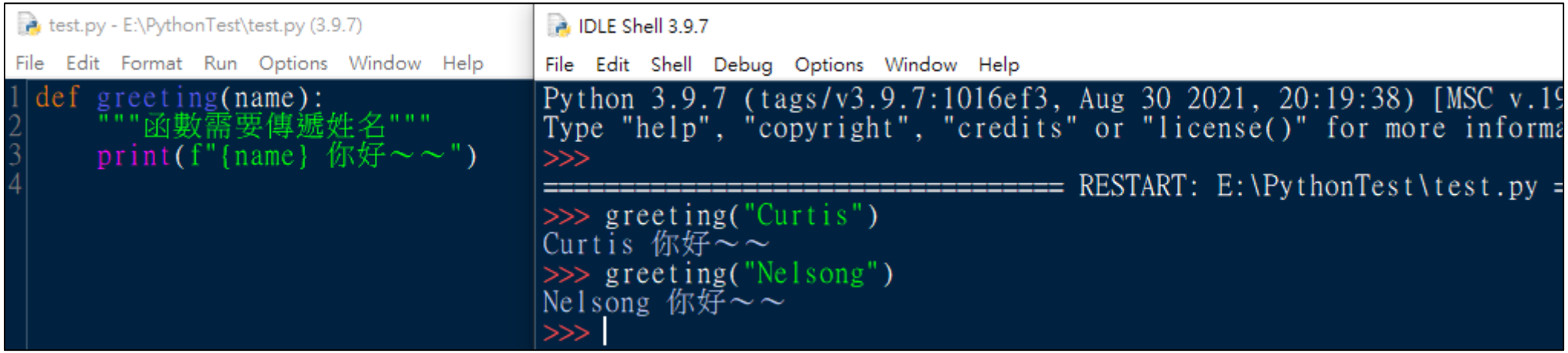


11-2 函數的參數設計

- 實際上的函數設計大多都要傳遞參數。
  - print(), max(), min(), …

```
1   def greeting(name):
2       """函數需要傳遞姓名"""
3       print(f"{name} 你好～～")
4
5   greeting("Curtis")
```

```
Curtis 你好～～
```



11-2 函數的參數設計

- 多個參數的傳遞
  - 傳遞參數的位置/順序需要正確
  - 參數最常見是數值或字串
  - 有時也會使用串列、元組、字典或是函數

```
1   def subtract(x1, x2):
2       """減法函數"""
3       result = x1 - x2
4       print(f"{x1} - {x2} = {result}")
5
6   a = 100
7   b = 50
8
9   print(f"{a} - {b} = {a-b}")
10  subtract(a, b)
11  subtract(b, a)  #參數給反了
```

```
100 - 50 = 50
100 - 50 = 50
50 - 100 = -50
```

11-2 函數的參數設計

- 關鍵字參數：參數名稱=值
  - 本質上是字典
  - 此時參數順序就不重要了

```python
def subtract(x1, x2):
    """減法函數"""
    result = x1 - x2
    print(f"{x1} - {x2} = {result}")

a = 100
b = 50

print(f"{a} - {b} = {a-b}")
subtract(a, b)
subtract(b, a)  #參數給反了

subtract(x2 = b, x1 = a)
```

```
100 - 50 = 50
100 - 50 = 50
50 - 100 = -50
100 - 50 = 50
```

11-2 函數的參數設計

- 設計函數時可以給參數預設值。
- 特別需留意：函數設計時含有預設值的參數，必需放置在參數列的最右邊。

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

11-2 函數的參數設計

- 設計函數時可以給參數預設值。
- 特別需留意：函數設計時含有預設值的參數，必需放置在參數列的最右邊。

```
1   def interest(interest_type, subject = "敦煌"):
2       """顯示興趣和主題"""
3       print(f"我的興趣是 {interest_type}")
4       print(f"在 {interest_type} 中，最喜歡的是 {subject}")
5       print()
6
7   interest(interest_type = "旅行")
8   interest("旅行", "洛杉磯")
9
10  interest(subject = "科學新知", interest_type = "讀書")
```

```
我的興趣是 旅行
在 旅行 中，最喜歡的是 敦煌

我的興趣是 旅行
在 旅行 中，最喜歡的是 洛杉磯

我的興趣是 讀書
在 讀書 中，最喜歡的是 科學新知
```

11-3 函數的回傳值

- 前面例子沒有寫回傳值，Python 會自動回傳 None。

```python
def greeting(name):
    """函數需要傳遞姓名"""
    print(f"{name} 你好~~")

retVal = greeting("Curtis")
print(f"greeting 的回傳值為 {retVal}")
print(f"greeting 回傳值的 type 為 {type(retVal)}")
```

```
Curtis 你好~~
greeting 的回傳值為 None
greeting 回傳值的 type 為 <class 'NoneType'>
```

```python
def greeting(name):
    """函數需要傳遞姓名"""
    print(f"{name} 你好~~")
    return

retVal = greeting("Curtis")
print(f"greeting 的回傳值為 {retVal}")
print(f"greeting 回傳值的 type 為 {type(retVal)}")
```

11-3 函數的回傳值

- None 的值可當作 bool 裡面的 False。

```python
def is_None(str, val):
    if val is None:
        print(f"{str} = None")
    elif val:   # val == True
        print(f"{str} = True")
    else:
        print(f"{str} = False")

is_None("空串列", [])
is_None("空元組", ())
is_None("空字典", {})
is_None("空集合", set())
is_None("None", None)
is_None("True", True)
is_None("False", False)
```

```
空串列 = False
空元組 = False
空字典 = False
空集合 = False
None = None
True = True
False = False
```

11-3 函數的回傳值

- 簡單回傳數值資料
  - return result　　#result 就是回傳的值

```
1   def subtract(x1, x2):
2       """減法函數"""
3       result = x1 - x2
4       return result
5
6   a = 100
7   b = 50
8
9   print(f"{a} - {b} = {subtract(a, b)}")
```

```
100 - 50 = 50
```

11-3 函數的回傳值

- 程式包含兩個函數的應用：

```
請輸入運算
 1: 加法
 2: 減法
請輸入 1/2: 1
a = 50
b = 60
50 + 60 = 110
```

```python
def subtract(x1, x2):
    """減法函數"""
    return x1 - x2

def addition(x1, x2):
    """加法函數"""
    return x1 + x2

print("請輸入運算\n", "1: 加法\n", "2: 減法")

op = input("請輸入 1/2: ")
a = int(input("a = "))
b = int(input("b = "))

if(op == "1"):
    print(f"{a} + {b} = {addition(a, b)}")
elif (op == "2"):
    print(f"{a} - {b} = {subtract(a, b)}")
else:
    print("運算方式輸入錯誤！！！")
```

11-3 函數的回傳值

- 傳回多筆資料的應用 – 實際上是回傳 tuple
  - return 多筆資料，資料與資料間用 "," 隔開。

```
1   def arithFunction(x1, x2):
2       """加減乘除四則運算的結果"""
3       addRes = x1 + x2
4       subRes = x1 - x2
5       mulRes = x1 * x2
6       divRes = x1 / x2
7       return addRes, subRes, mulRes, divRes
8
9   x = 50
10  y=  100
11
12  #add, sub, mul, div = arithFunction(x, y)
13  ans = arithFunction(x, y)
14  print(f"{type(ans) = }")
15  print(f"加法結果 = {ans[0]}")
16  print(f"減法結果 = {ans[1]}")
17  print(f"乘法結果 = {ans[2]}")
18  print(f"除法結果 = {ans[3]}")
```

Unpack tuple

```
type(ans) = <class 'tuple'>
加法結果 = 150
減法結果 = -50
乘法結果 = 5000
除法結果 = 0.5
```

11-3 函數的回傳值

- 傳回多筆資料的應用 – 實際上是回傳 list
  - return 多筆資料，資料與資料間用 "," 隔開。

```
1   def arithFunction(x1, x2):
2       """加減乘除四則運算的結果"""
3       addRes = x1 + x2
4       subRes = x1 - x2
5       mulRes = x1 * x2
6       divRes = x1 / x2
7       return [addRes, subRes, mulRes, divRes]
8
9   x = 50
10  y=  100
11
12  #add, sub, mul, div = arithFunction(x, y)
13  ans = arithFunction(x, y)
14  print(f"{type(ans) = }")
15  print(f"加法結果 = {ans[0]}")
16  print(f"減法結果 = {ans[1]}")
17  print(f"乘法結果 = {ans[2]}")
18  print(f"除法結果 = {ans[3]}")
```

Unpack list

```
type(ans) = <class 'list'>
加法結果 = 150
減法結果 = -50
乘法結果 = 5000
除法結果 = 0.5
```

11-3 函數的回傳值

- 傳回字串的應用：

```python
def arithFunction(x1, x2, op = "+"):
    """加減乘除四則運算的結果"""
    if op == "+":
        opRes = f"{x1} + {x2} = {x1 + x2}"
    elif op == "-":
        opRes = f"{x1} - {x2} = {x1 - x2}"
    elif op == "*":
        opRes = f"{x1} * {x2} = {x1 * x2}"
    elif op == "/":
        opRes = f"{x1} / {x2} = {x1 / x2}"
    else:
        opRes = "沒有定義此運算"
    return opRes

x, y = 50, 100

print(arithFunction(x, y))
print(arithFunction(x, y, "-"))
print(arithFunction(x, y, "*"))
print(arithFunction(x, y, "/"))
print(arithFunction(x, y, "^"))
```

```
50 + 100 = 150
50 - 100 = -50
50 * 100 = 5000
50 / 100 = 0.5
沒有定義此運算
```

11-3 函數的回傳值

- 傳回字典的應用：

```python
def build_VIP(id, name, tel =""):
    """建立 VIP 資訊"""
    dictVIP = {"VIP_ID":id, "Name":name}
    if tel:
        dictVIP["Tel"] = tel
    return dictVIP


dictVIP1 = build_VIP(1, "Curtis")
dictVIP2 = build_VIP(2, "Nelson", "0912345678")
print(dictVIP1)
print(dictVIP2)
```

```
{'VIP_ID': 1, 'Name': 'Curtis'}
{'VIP_ID': 2, 'Name': 'Nelson', 'Tel': '0912345678'}
```

11-3 函數的回傳值

- 傳回字典的應用(改)：

```
1   def build_VIP(dict, id, name, tel =""):
2       """建立 VIP 資訊"""
3       dict[id] = (name, tel)
4
5   dictVIP = {}
6
7   build_VIP(dictVIP, 1, "Curtis")
8   print(dictVIP)
9   build_VIP(dictVIP, 2, "Nelson", "0912345678")
10  print(dictVIP)
```

```
{1: ('Curtis', '')}
{1: ('Curtis', ''), 2: ('Nelson', '0912345678')}
```

10.4 呼叫函數參數是串列

- 在呼叫函數時，也可以將串列(此串列可以是由數值、字串或字典所組成)當參數傳遞給函數

```python
def CourseMsg(students):
    str1 = " 同學好，"
    str2 = "期中考於 11/9 在電腦教室舉行上機考"
    str3 = "老師 Curtis"
    for student in students:
        print(f"{student}{str1}\n{str2}\n{str3}\n")

students = ("Nelson", "Jason", "Wilson")
CourseMsg(students)
```

```
Nelson 同學好，
期中考於 11/9 在電腦教室舉行上機考
老師 Curtis

Jason 同學好，
期中考於 11/9 在電腦教室舉行上機考
老師 Curtis

Wilson 同學好，
期中考於 11/9 在電腦教室舉行上機考
老師 Curtis
```

10.4 呼叫函數參數是串列

- 觀察傳遞一般變數與串列變數到函數的區別
  - 傳遞整數資料

```
1   def IntData(n):
2       print(f"副程式 {id(n)=}, {n=}")
3       n = 5
4       print(f"副程式 {id(n)=}, {n=}")
5
6   x = 1
7   print(f"主程式 {id(x)=}, {x=}")
8   IntData(x)
9   print(f"主程式 {id(x)=}, {x=}")
```

```
主程式 id(x)=2084704944368, x=1
副程式 id(n)=2084704944368, n=1
副程式 id(n)=2084704944496, n=5
主程式 id(x)=2084704944368, x=1
```

  - 傳遞串列資料

```
1   def IntData(n):
2       print(f"副程式 {id(n)=}, {n=}, {id(n[0])=}")
3       n[0] = 5
4       print(f"副程式 {id(n)=}, {n=}, {id(n[0])=}")
5
6   x = [1, 2]
7   print(f"主程式 {id(x)=}, {x=}, {id(x[0])=}")
8   IntData(x)
9   print(f"主程式 {id(x)=}, {x=}, {id(x[0])=}")
```

```
主程式 id(x)=2019451682752, x=[1, 2], id(x[0])=2019446423792
副程式 id(n)=2019451682752, n=[1, 2], id(n[0])=2019446423792
副程式 id(n)=2019451682752, n=[5, 2], id(n[0])=2019446423920
主程式 id(x)=2019451682752, x=[5, 2], id(x[0])=2019446423920
```

10.4 呼叫函數參數是串列

- 在函數內修訂串列的內容
  - 設計一個點餐系統，顧客在點餐時，可以將所點的餐點放入unserved串列，服務完成後將已服務餐點放入served串列。

```python
def show_unserved_meal(unserved):
    """顯示尚未服務的餐點"""
    print("=== 下列是未出餐的餐點 ===")
    if unserved:
        for meal in unserved:
            print(f"{meal}")
    else:
        print("*** 沒有餐點 ***\n")

def show_served_meal(served):
    """顯示已服務的餐點"""
    print("=== 下列是已出餐的餐點 ===")
    if served:
        for meal in served:
            print(f"{meal}")
    else:
        print("*** 沒有餐點 ***\n")

def kitchen(unserved, served):
    """將未服務餐點轉為已服務餐點"""
    print("廚房處理顧客所點的餐點")
    while unserved:
        currentMeal = unserved.pop()
        #模擬出餐過程
        print(f"出餐：{currentMeal}")
        served.append(currentMeal)

unserved = ["大麥克", "勁辣雞腿堡", "麥脆雞"]    #已點餐點
served = []                                      #已服務餐點

#列出廚房處理前的餐點內容
show_unserved_meal(unserved)
show_served_meal(served)

#廚房處理餐點過程
kitchen(unserved, served)
print("\n=== 廚房處理結束 ===\n")

#列出廚房處理後的餐點內容
show_unserved_meal(unserved)
show_served_meal(served)
```

變數名稱與參數名稱不見得要一樣

```
=== 下列是未出餐的餐點 ===
大麥克
勁辣雞腿堡
麥脆雞
=== 下列是已出餐的餐點 ===
*** 沒有餐點 ***

廚房處理顧客所點的餐點
出餐：麥脆雞
出餐：勁辣雞腿堡
出餐：大麥克

=== 廚房處理結束 ===

=== 下列是未出餐的餐點 ===
*** 沒有餐點 ***

=== 下列是已出餐的餐點 ===
麥脆雞
勁辣雞腿堡
大麥克
```

10.4 呼叫函數參數是串列

- 在函數內修訂串列的內容
  - 設計一個點餐系統，顧客在點餐時，可以將所點的餐點放入unserved串列，服務完成後將已服務餐點放入served串列。

```python
def show_unserved_meal(unserved):
    """顯示尚未服務的餐點"""
    print("=== 下列是未出餐的餐點 ===")
    if unserved:
        for meal in unserved:
            print(f"{meal}")
    else:
        print("*** 沒有餐點 ***\n")

def show_served_meal(served):
    """顯示已服務的餐點"""
    print("=== 下列是已出餐的餐點 ===")
    if served:
        for meal in served:
            print(f"{meal}")
    else:
        print("*** 沒有餐點 ***\n")

def kitchen(unserved, served):
    """將未服務餐點轉為已服務餐點"""
    print("廚房處理顧客所點的餐點")
    while unserved:
        currentMeal = unserved.pop()
        #模擬出餐過程
        print(f"出餐：{currentMeal}")
        served.append(currentMeal)

unserved = ["大麥克", "勁辣雞腿堡", "麥脆雞"]    #已點餐點
served = []                                      #已服務餐點

#列出廚房處理前的餐點內容
show_unserved_meal(unserved)
show_served_meal(served)

#廚房處理餐點過程
kitchen(unserved[:], served)
print("\n=== 廚房處理結束 ===\n")

#列出廚房處理後的餐點內容
show_unserved_meal(unserved)
show_served_meal(served)
```

```
=== 下列是未出餐的餐點 ===
大麥克
勁辣雞腿堡
麥脆雞
=== 下列是已出餐的餐點 ===
*** 沒有餐點 ***

廚房處理顧客所點的餐點
出餐：麥脆雞
出餐：勁辣雞腿堡
出餐：大麥克

=== 廚房處理結束 ===

=== 下列是未出餐的餐點 ===
大麥克
勁辣雞腿堡
麥脆雞
=== 下列是已出餐的餐點 ===
麥脆雞
勁辣雞腿堡
大麥克
```

10.4 呼叫函數參數是串列

- 傳遞串列的提醒
  - 假設參數串列的預設值是空串列，在重複呼叫過程預設串列會遺留先前呼叫的內容。

```
1   def insertChar(char, myList=[], inList=[1, 2]):
2       print(f"{id(myList) = }, {myList = }")
3       print(f"{id(inList) = }, {inList = }")
4       myList.append(char)
5       inList.append(char)
6       print(f"{id(myList) = }, {myList = }")
7       print(f"{id(inList) = }, {inList = }")
8
9   insertChar("x")
10  insertChar("y")
```

```
id(myList) = 2256540811200, myList = []
id(inList) = 2256540744896, inList = [1, 2]
id(myList) = 2256540811200, myList = ['x']
id(inList) = 2256540744896, inList = [1, 2, 'x']
id(myList) = 2256540811200, myList = ['x']
id(inList) = 2256540744896, inList = [1, 2, 'x']
id(myList) = 2256540811200, myList = ['x', 'y']
id(inList) = 2256540744896, inList = [1, 2, 'x', 'y']
```

10.4 呼叫函數參數是串列

- 傳遞串列的提醒
  - 假設參數串列的預設值是空串列，在重複呼叫過程預設串列會遺留先前呼叫的內容。
  - 避免這種情況，將串列參數預設值改為 None

```
1   def insertChar(char, myList=None, inList=[]):
2       print(f"{id(myList) = }, {myList = }")
3       print(f"{id(inList) = }, {inList = }")
4       if( myList == None):
5           myList = []
6       myList.append(char)
7       inList.append(char)
8       print(f"{id(myList) = }, {myList = }")
9       print(f"{id(inList) = }, {inList = }")
10
11  insertChar("x")
12  insertChar("y")
```

```
id(myList) = 140717570496504, myList = None
id(inList) = 2361657868224, inList = []
id(myList) = 2361657801920, myList = ['x']
id(inList) = 2361657868224, inList = ['x']
id(myList) = 140717570496504, myList = None
id(inList) = 2361657868224, inList = ['x']
id(myList) = 2361657801920, myList = ['y']
id(inList) = 2361657868224, inList = ['x', 'y']
```

11-5：傳遞任意數量的參數

- 有時候會碰上不知道會有多少個參數會傳遞到這個函數，此時可以將參數前面加上“*”，表示可以有 0 到多個參數將傳遞到此函數內。
- 此時參數會被群組化為元組 (tuple)。

```
1   def makeIceCream(*toppings):
2       """列出製作冰淇淋的配料"""
3       print(f"此冰淇淋的配料如下：")
4       for topping in toppings:
5           print(f"---- {topping}")
6       print(f"{type(toppings)}")
7       print(f"{toppings}")
8
9   makeIceCream("草莓果醬")
10  makeIceCream("草莓果醬", "巧克力醬", "新鮮草莓")
```

```
此冰淇淋的配料如下：
---- 草莓果醬
<class 'tuple'>
('草莓果醬',)
此冰淇淋的配料如下：
---- 草莓果醬
---- 巧克力醬
---- 新鮮草莓
<class 'tuple'>
('草莓果醬', '巧克力醬', '新鮮草莓')
```

11-5：傳遞任意數量的參數

- 遇上一般參數與任意數量參數的搭配時，任意數量的參數必須放在最右邊。

```
1   def makeIceCream(iceCream, *toppings):
2       """列出製作冰淇淋的配料"""
3       print(f"{iceCream} 冰淇淋的配料如下：")
4       for topping in toppings:
5           print(f"---- {topping}")
6
7   makeIceCream("草莓", "草莓果醬")
8   makeIceCream("草莓巧克力", "草莓果醬", "巧克力醬", "新鮮草莓")
```

```
草莓 冰淇淋的配料如下：
---- 草莓果醬
草莓巧克力 冰淇淋的配料如下：
---- 草莓果醬
---- 巧克力醬
---- 新鮮草莓
```

11-5：傳遞任意數量的參數

- 任意數量的關鍵字參數
  - 在參數前面加上 "**" 。
  - 參數會變成任意數量的字典元素，其中引數是鍵值，對應的值是字典的值。

```
1   def build_dict(name, age, **players):
2       """建立 NBA 球員的字典資料"""
3       info = {}   #建立空字典
4       info["Name"] = name
5       info["Age"] = age
6       for key, value in players.items():
7           info[key] = value
8       return info
9
10  palyer_dict = build_dict("James", 32, City = "Cleveland", State = "Ohio")
11  print(palyer_dict)
```

```
{'Name': 'James', 'Age': 32, 'City': 'Cleveland', 'State': 'Ohio'}
```

11-6：進一步認識函數

```
1   def greeting(name):
2       """函數需要傳遞姓名"""
3       print(f"{name} 你好～～")
4
5   greeting("Curtis")
```

文件字串 docstring (document string 的縮寫)

- 用 help 函數列出此函數的文件字串

11-6：進一步認識函數

- 函數也是個物件

```python
def upperStr(text):
    return text.upper()

print(upperStr("Python"))

upperLetter = upperStr
print(upperLetter("Python"))

print(f"{type(upperStr) = }")
print(f"{type(upperLetter) = }")

print(f"{id(upperStr) = }")
print(f"{id(upperLetter) = }")
```

```
PYTHON
PYTHON
type(upperStr) = <class 'function'>
type(upperLetter) = <class 'function'>
id(upperStr) = 2853190516256
id(upperLetter) = 2853190516256
```

11-6：進一步認識函數

- 函數可以是資料結構成員

```
<built-in function max> 10
<built-in function min> 1
<built-in function sum> 16
<function summation at 0x0000020CBBB33E20> 16
```

```python
def summation(data):
    return sum(data)

x = (1, 5, 10)
funcList = (max, min, sum, summation)
for f in funcList:
    print(f, f(x))
```

- 函數可以當作參數傳給其他函數

```
resAdd = 30
resMul = 200
```

```python
def add(x, y):
    return x + y

def mul(x, y):
    return x * y

def runFunc(func, arg1, arg2):
    return func(arg1, arg2)

resAdd = runFunc(add, 10, 20)
resMul = runFunc(mul, 10, 20)

print(f"{resAdd = }\n{resMul = }")
```

11-6：進一步認識函數

- 函數當參數與*args不定量的參數

```
1   def mySum(*data):
2       return sum(data)
3
4   def runWithMiltipleData(func, *data):
5       print(data)
6       print(*data)
7       return func(*data)
8
9   print(runWithMiltipleData(mySum, 1, 2, 3, 4, 5))
10  print(runWithMiltipleData(mySum, 6, 7, 8, 9, 10))
```

注意 * 號

```
(1, 2, 3, 4, 5)
1 2 3 4 5
15
(6, 7, 8, 9, 10)
6 7 8 9 10
40
```

11-6：進一步認識函數

- 嵌套函數
  - 函數內部也可以有函數，有時候可以利用這個特性執行複雜的運算。
  - 也具有可重複使用、封裝，隱藏數據的效果。

```
1   def dist(x1, y1, x2, y2):
2       def mySqrt(z):
3           return z ** 0.5
4       dx = (x1 - x2) ** 2
5       dy = (y1 - y2) ** 2
6       return mySqrt(dx + dy)
7
8   print(dist(0, 0, 1, 2))
```

```
2.23606797749979
```

11-6：進一步認識函數

- 嵌套函數
  - 函數也可以當作回傳值。

```
1   def outer():
2       def inner(n):
3           print("inner running")
4           return sum(range(n))
5       return inner
6
7   f = outer()
8   print(f)
9   print(f(5))
10
11  y = outer()
12  print(y)
13  print(y(10))
```

```
<function outer.<locals>.inner at 0x0000021608A9DAB0>
inner running
10
<function outer.<locals>.inner at 0x0000021608A9DA20>
inner running
45
```

11-6：進一步認識函數

- 閉包 closure
  - 內部函數是個動態產生的程式，可以記住函數以外的程式所建立的環境變數的值時，我們可以稱這個內部函數是閉包closure

```
1   def outer(a, b):
2       """參數 a 跟 b 將是 inner 的環境變數"""
3       a, b = 5, 10
4       def inner(x):
5           return a * x + b
6       return inner
7
8   b = 20
9   f = outer(15, 10)
10  print(f(b))
11
12  print(f)
13  print(f.__closure__)
14  print(f.__closure__[0].cell_contents)
15  print(f.__closure__[1].cell_contents)
```

```
110
<function outer.<locals>.inner at 0x000002904792DAB0>
(<cell at 0x0000029047741000: int object at 0x00000290475B0170>,
<cell at 0x00000290477419F0: int object at 0x00000290475B0210>)
5
10
```

- a, b 和 inner() 形成 closure。
- __closure__ 是一個元組，環境變數存在 cell_contents 內。

11-6：進一步認識函數

- 閉包 closure 的一個應用
  - 動態產生線性函數

```
1   def outer(a, b):
2       """參數 a 跟 b 將是 inner 的環境變數"""
3       def inner(x):
4           return a * x + b
5       return inner
6
7   b = 20
8   f = outer(5, 10)
9   print(f"{f(b) = }")
10
11  y = outer(10, 5)
12  print(f"{y(b) = }")
```

```
f(b) = 110
y(b) = 205
```

  - 讓程式碼更有彈性，擴充性更好。

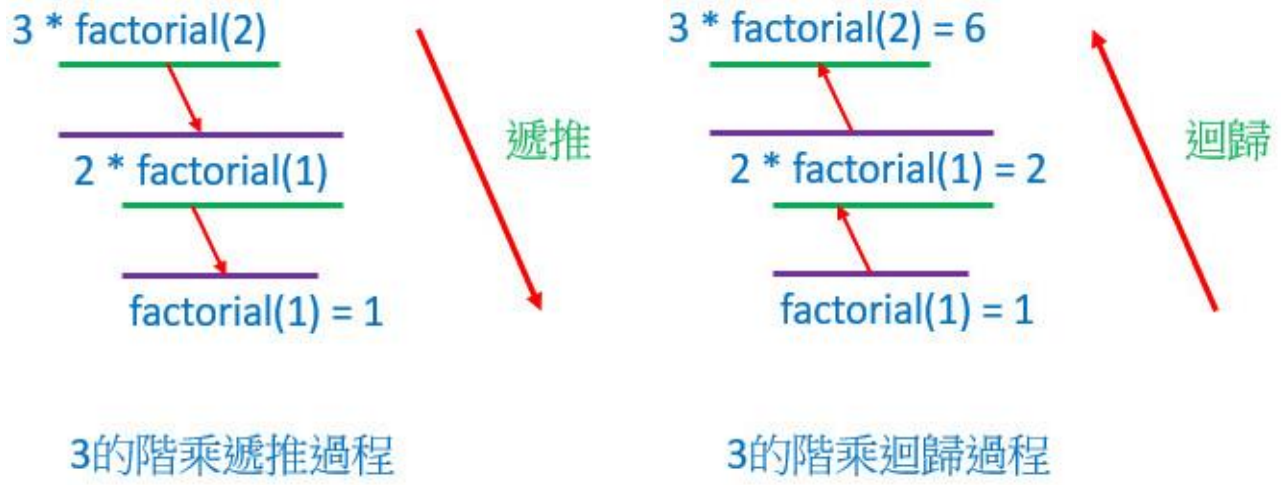
11-7：遞迴式函數設計recursive

- 一個函數可以呼叫其它函數也可以呼叫自己，其中呼叫本身的動作稱遞迴式(recursive)呼叫。
  - 每次呼叫自己時，都會使範圍越來越小。
  - 必需要有一個終止的條件來結束遞迴函數。

```python
def factorial(n):
    """計算 n 的階乘"""
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

print(f"3 階乘的結果是 {factorial(3)}")
print(f"5 階乘的結果是 {factorial(5)}")
```

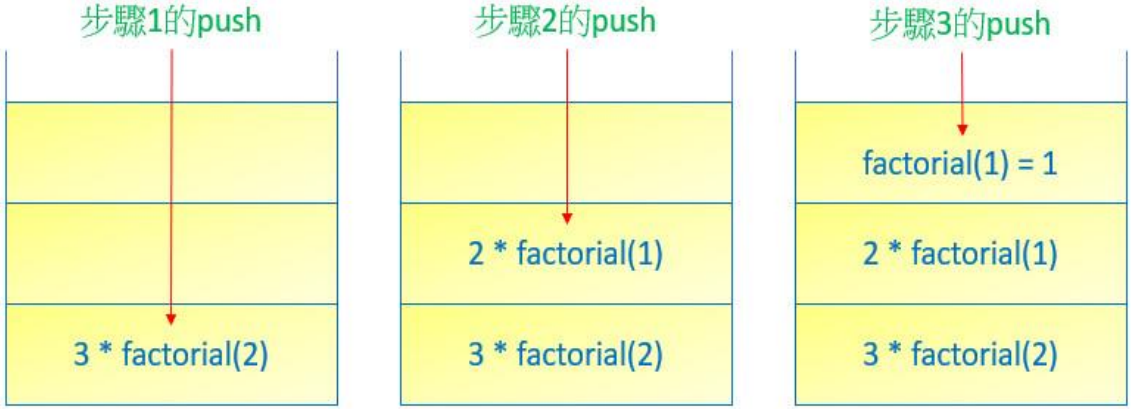
```
3 階乘的結果是 6
5 階乘的結果是 120
```



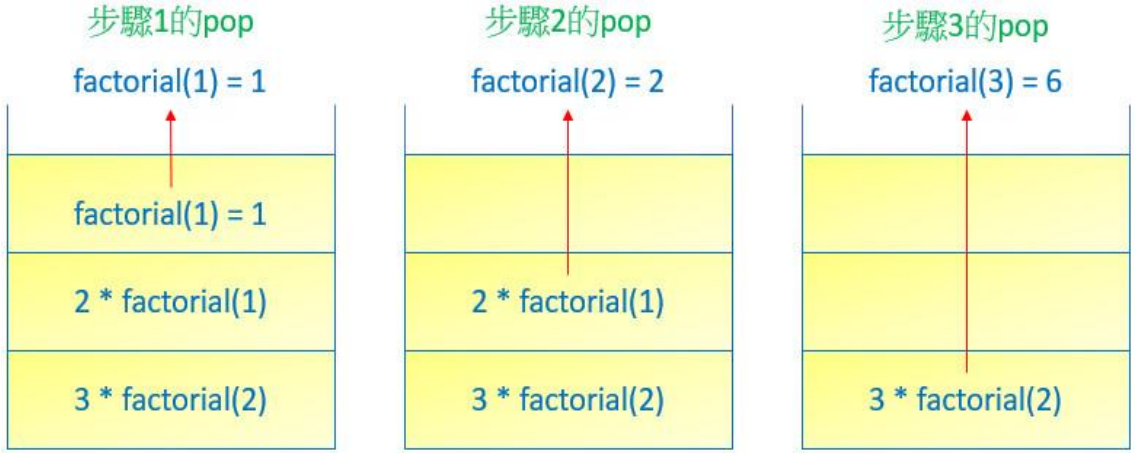
3的階乘遞推過程　　3的階乘迴歸過程

11-7：遞迴式函數設計recursive

- 呼叫函數時，會使用堆疊(stack):
  - 後進先出
  - push跟 pop



階乘計算使用堆疊(stack)的說明，這是由左到右進入堆疊push操作過程



階乘計算使用堆疊(stack)的說明，這是由左到右離開堆疊的pop過程

- Python 預設最多大遞迴次數為 1000，可透過 sys.getrecursionlimit() 得到目前遞迴的最大次數。透過 sys.setrecursionlimit() 設定最大遞迴次數。

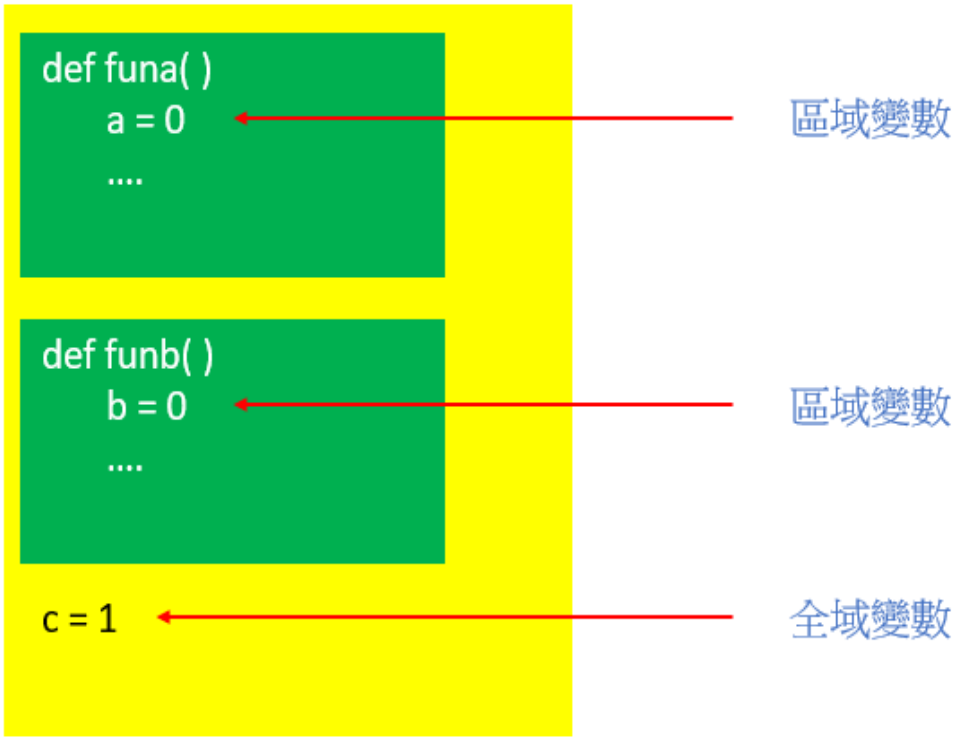
```
>>> import sys
>>> sys.getrecursionlimit()
1000
```

11-8：區域變數與全域變數

- 影響範圍限定在這個函數內，這個變數稱區域變數(local variable)。如果某個變數的影響範圍是在整個程式，則這個變數稱全域變數(global variable)。

```
1   def printmsg():
2       print(f"printmsg: {msg}")
3
4   msg = "Global msg variable"
5   print(f"main: {msg}")
6   printmsg()
```

```
main: Global msg variable
printmsg: Global msg variable
```



11-8：區域變數與全域變數

- 區域變數與全域變數使用相同的名稱:
  - 將相同名稱的區域與全域變數視為不同的變數
  - 區域變數所在的函數使用區域變數內容
  - 其他區域使用全域變數的內容

```python
def printmsgL():
    msg = "Local msg variable"
    print(f"printmsgL: {msg}")

def printmsgG():
    print(f"printmsgG: {msg}")

msg = "Global msg variable"
print(f"main: {msg}")
printmsgL()
printmsgG()
```

```
main: Global msg variable
printmsgL: Local msg variable
printmsgG: Global msg variable
```

11-8：區域變數與全域變數

- 區域變數內容無法在其它函數引用。
- 區域變數內容無法在主程式引用。
- 在函數內不能更改全域變數的值。
- 如果要在函數內要修改全域變數值，需在函數內使用global 宣告此變數。

```python
def printmsgL():
    msg = "printmsgL msg variable"
    print(f"printmsgL: {msg}")

def printmsgG():
    global msg
    msg = "printmsgG msg variable"
    print(f"printmsgG: {msg}")

msg = "Global msg variable"
print(f"main: {msg}")
printmsgL()
printmsgG()
print(f"main: {msg}")
```

```
main: Global msg variable
printmsgL: printmsgL msg variable
printmsgG: printmsgG msg variable
main: printmsgG msg variable
```

11-8：區域變數與全域變數

- locals( )：可以用字典方式列出所有的區域變數名稱與內容。
- globals( )：可以用字典方式列出所有的全域變數名稱與內容。

```
1   def printLocal():
2       """印出區域變數"""
3       lang = "Java"
4       print("語言：", lang)
5       print("區域變數：", locals())
6
7   msg = "Python"
8   printLocal()
9   print("語言：", msg)
10  print("全域變數：", globals())
```

```
語言： Java
區域變數： {'lang': 'Java'}
語言： Python
全域變數： {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.Sourc
eFileLoader object at 0x00000193D665C310>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (b
uilt-in)>, '__file__': 'e:\\PythonTest\\test.py', '__cached__': None, 'printLocal': <function printLocal at 0x00000193D65
6F1F0>, 'msg': 'Python'}
```

11-8：區域變數與全域變數

- nonlocal變數
  - 用法與 global相同。
  - global 是指最上層 (main) 的變數。
  - nonlocal 指的是上一/幾層的變數。

```
main 輸出 var_global = 1
main 輸出 var_nonlocal = 2
local_fun 輸出 var_global = 111
local_fun 輸出 var_nonlocal = 222
main 輸出 var_global = 111
main 輸出 var_nonlocal = 2
```

```python
def local_func():
    var_nonlocal = 22
    def local_innerFunc():
        global var_global
        nonlocal var_nonlocal
        var_global = 111
        var_nonlocal = 222
    local_innerFunc()
    print(f"local_fun 輸出 {var_global = }")
    print(f"local_fun 輸出 {var_nonlocal = }")

var_global = 1
var_nonlocal = 2
print(f"main 輸出 {var_global = }")
print(f"main 輸出 {var_nonlocal = }")
local_func()
print(f"main 輸出 {var_global = }")
print(f"main 輸出 {var_nonlocal = }")
```

11-9：匿名函數lambda

- 所謂的匿名函數(anonymous function)是指一個沒有名稱的函數，適合使用在程式中只存在一小段時間的情況。
- Python是使用def定義一般函數，匿名函數則是使用lambda來定義，有的人稱之為lambda表達式，也可以將匿名函數稱lambda函數。
- 有時會與內建的 filter(), map(), reduce(), … 等共同使用。
- 語法：
  - lambda arg1[, arg2, … arg*n*]:expression　# arg1是參數，可以有多個參數

```
#def square(x):
#    return x ** 2

square = lambda x: x ** 2

print(square(10))
```

```
100
```

```
mul2 = lambda x, y: x * y

print(mul2(10, 20))
```

```
200
```

11-9：匿名函數lambda

- 使用lambda匿名函數的理由
  - 較佳的使用時機是存在一個函數的內部（回想嵌套函數）

```
1   def outer(a, b):
2       """參數 a 跟 b 將是 inner 的環境變數"""
3       #def inner(x):
4       #    return a * x + b
5       #return inner
6       return lambda x: a * x + b
7
8   b = 20
9   f = outer(5, 10)
10  print(f"{f(b) = }")
11
12  y = outer(10, 5)
13  print(f"{y(b) = }")
```

```
f(b) = 110
y(b) = 205
```

11-9：匿名函數lambda

- 匿名函數應用在高階函數的參數
  - 匿名函數一般是用在不需要函數名稱的場合。
  - 一些高階函數(Higher-order function)的部分參數是函數，這時就很適合使用匿名函數，同時讓程式變得更簡潔。

```python
def myCar(cars, func):
    for car in cars:
        print(func(car))

def wdcar(carBrand):
    return f"My dream car is {carBrand.title()}"

dreamCars = ["porsche", "rolls royce", "ferrari"]
myCar(dreamCars, wdcar)
```

```
My dream car is Porsche
My dream car is Rolls Royce
My dream car is Ferrari
```

```python
def myCar(cars, func):
    for car in cars:
        print(func(car))

dreamCars = ["porsche", "rolls royce", "ferrari"]
myCar(dreamCars, lambda carBrand: f"My dream car is {carBrand.title()}")
```

11-9：匿名函數lambda

- 匿名函數使用與 filter( )
  - filter(): 內建函數，用來篩選序列，語法格式如下：
    - filter(func, iterable)
    - 將 iterable 的元素 (item) 放進 func 內，然後將 func()函數執行結果為 True 的元素組成新的篩選物件 (filter object) 後回傳。

```
1   def oddFunc(x):
2       return x if(x % 2 == 1) else None
3
4   myList = [5, 10, 15, 20, 25, 30]
5   filter_obj = filter(oddFunc, myList)
6
7   print(f"奇數串列為：{filter_obj}")
8
9   print(f"奇數串列為：{[item for item in filter_obj]}")
10  #print(f"奇數串列為：{list(filter_obj)}")
```

```
<class 'filter'>
奇數串列為：<filter object at 0x000002781B281400>
奇數串列為：[5, 15, 25]
```

```
1   myList = [5, 10, 15, 20, 25, 30]
2   oddList = list(filter(lambda x: (x % 2 == 1), myList))
3   print(f"奇數串列為：{oddList}")
```

```
奇數串列為：[5, 15, 25]
```

11-9：匿名函數lambda

- 匿名函數使用與 map( )
  - map(): 內建函數，語法格式如下：
    - map(func, iterable)
    - 將 iterable 的元素 (item) 放進 func 內，然後將 func()函數執行結果回傳。

```
1   myList = [5, 10, 15, 20, 25, 30]
2   squareList = list(map(lambda x: (x ** 2), myList))
3   print(f"串列平方為：{squareList}")
```

```
串列平方為：[25, 100, 225, 400, 625, 900]
<class 'map'>
```

11-9：匿名函數lambda

- 匿名函數使用與 reduce( )
  - reduce(): 內建函數，語法格式如下：
    - reduce(func, iterable)
    - 先對可迭代物件的第1和第2個元素操作，結果再和第3個元素操作，直到最後一個元素。
    - 假設iterable有4個元素，可以用下列方式解說：
      - reduce(f, [a, b, c, d]) = f(f(f(a, b), c), d)
  - 早期 reduce 是內建函數，現被移至 functools 模組，所以使用前要 import functools

11-9：匿名函數lambda

- 匿名函數使用與 reduce( )

```python
from functools import reduce

def strToInt(s):
    def func(x, y):
        return 10 * x + y
    def charToNum(s):
        print(f"s = {type(s)} {s}")
        myDict = {"0":0, "1":1, "2":2, "3":3, "4":4, "5":5, "6":6, "7":7, "8":8, "9":9}
        n = myDict[s]
        print(f"n = {type(n)} {n}")
        return n
    return reduce(func, map(charToNum, s))

string = "8527"
x = strToInt(string) + 1000
print(f"{x = }")
```

```
s = <class 'str'> 8
n = <class 'int'> 8
s = <class 'str'> 5
n = <class 'int'> 5
s = <class 'str'> 2
n = <class 'int'> 2
s = <class 'str'> 7
n = <class 'int'> 7
x = 9527
```

產生 [8, 5, 2, 7]

```python
from functools import reduce

def strToInt(s):
    def charToNum(s):
        return {"0":0, "1":1, "2":2, "3":3, "4":4, "5":5, "6":6, "7":7, "8":8, "9":9}[s]
    return reduce(lambda x, y: 10 * x + y, map(charToNum, s))

string = "8527"
x = strToInt(string) + 1000
print(f"{x = }")
```

11-10：pass與函數 / 11-11：type關鍵字應用在函數

- pass: 第七章有介紹到
  - 大型程式可以先將各個函數做規劃，然後再一一完成各個函數。
  - 可先將未完成的函數加上 pass，讓 Python 在執行時不要報錯。

```python
from functools import reduce

def func(arg1, arg2):
    pass

myList = [1, 2, 3]

print(f"func 的 type: {type(func)}")
print(f"lambda 的 type: {type(lambda x: x)}")
print(f"內建函數 abs 的 type: {type(abs)}")

print(f"內建函數 filter 回傳的 type: {type(filter(func, myList))}")
print(f"內建函數 map 回傳的 type: {type(map(func, myList))}")
print(f"內建函數 reduce 回傳的 type: {type(reduce(func, myList))}")
```

```
func 的 type: <class 'function'>
lambda 的 type: <class 'function'>
內建函數 abs 的 type: <class 'builtin_function_or_method'>
內建函數 filter 回傳的 type: <class 'filter'>
內建函數 map 回傳的 type: <class 'map'>
內建函數 reduce 回傳的 type: <class 'NoneType'>
```

11-12：設計自己的range( )

- 在 Python 2.x，range 回傳的是串列。
- 在 Python 3.x，range 回傳的是 range 物件。
  - 最大特色是不用預先儲存所有序列範圍的值，可以節省記憶體以及增加程式效率。
  - 每次迭代，會記得上次呼叫的位置，同時回傳下一個位置。

```
1   def myRange(start = 0, stop = 100, step = 1):
2       n = start
3       while n < stop:
4           yield n
5           n += step
6
7   print(type(myRange))
8   for x in myRange(0, 10, 2):
9       print(x)
```

```
<class 'function'>
0
2
4
6
8
```

先將此時的 n 回傳，下一次迭代繼續時，會從下一行繼續執行。

  - 將 range() 此觀念稱為生成器 (generator)

11-13：裝飾器(Decorator)

- 裝飾器 (decorator)
  - 在一些原有函數內增加一些功能，但是不希望修改到原本函數。
  - 本質上也是一個函數，會接收一個函數，然後回傳另一個函數。
  - Ex:

```
1  def greeting(string):
2      return string
3
4  print(greeting("Hello Python!!!"))
```

```
Hello Python!!!
```

  - 使用修飾器將輸出改為全大寫字串：

```
1  def upperGreeting(func):
2      def newFunc(args):
3          oldRes = func(args)
4          newRes = oldRes.upper()
5          print(f"函數名稱：{func.__name__}")
6          print(f"函數參數：{args}")
7          return newRes
8      return newFunc
9
10 def greeting(string):
11     return string
12
13 print(greeting("Hello Python!!!"))
14 myGreeting = upperGreeting(greeting)
15 print(myGreeting("Hello Python!!!"))
```

```
Hello Python!!!
函數名稱：greeting
函數參數：Hello Python!!!
HELLO PYTHON!!!
```

11-13：裝飾器(Decorator)

- Python 提供裝飾器的簡易設定方式
  - 使用 "@decorator"，decorator 是裝飾器的名稱。

```python
def upperGreeting(func):
    def newFunc(args):
        oldRes = func(args)
        newRes = oldRes.upper()
        print(f"函數名稱：{func.__name__}")
        print(f"函數參數：{args}")
        return newRes
    return newFunc

@upperGreeting
def greeting(string):
    return string

print(greeting("Hello Python!!!"))
```

```
函數名稱：greeting
函數參數：Hello Python!!!
HELLO PYTHON!!!
```

11-13：裝飾器(Decorator)

- 常用於為一個函數增加錯誤檢查的功能。

```
1   def myDiv(x, y):
2       return x/y
3
4   print(myDiv(3, 4))
5   print(myDiv(3, 0))
```

```
0.75
Traceback (most recent call last):
  File "e:\PythonTest\test.py", line 5, in <module>
    print(myDiv(3, 0))
  File "e:\PythonTest\test.py", line 2, in myDiv
    return x/y
ZeroDivisionError: division by zero
```

```
1   def errCheck(func):
2       def newFunc(*args):
3           if (args[1] != 0):
4               result = func(*args)
5           else:
6               result = "除數不可為 0"
7           print(f"函數名稱：{func.__name__}")
8           print(f"函數參數：{args}")
9           print(f"執行結果：{result}")
10          return result
11      return newFunc
12
13  @errCheck
14  def myDiv(x, y):
15      return x/y
16
17  print(myDiv(3, 4))
18  print(myDiv(3, 0))
```

```
函數名稱：myDiv
函數參數：(3, 4)
執行結果：0.75
0.75
函數名稱：myDiv
函數參數：(3, 0)
執行結果：除數不可為 0
除數不可為 0
```

11-13：裝飾器(Decorator)

- 一個函數可以有兩個以上的裝飾器

```python
def upperGreeting(func):
    def newFunc(args):
        print(f'This is function "upperGreeting".')
        oldRes = func(args)
        newRes = oldRes.upper()
        print(f"函數名稱：{func.__name__}")
        print(f"函數參數：{args}")
        return newRes
    return newFunc

def bold(func):
    def wrapper(args):
        print(f'This is function "bold".')
        return "bold-" + func(args) + "-bold"
    return wrapper

@bold
@upperGreeting
def greeting(string):
    return string

print(greeting("Hello Python!!!"))
```

執行順序為 bold -> upperGreeting

```
This is function "bold".
This is function "upperGreeting".
函數名稱：greeting
函數參數：Hello Python!!!
bold-HELLO PYTHON!!!-bold
```

11-13：裝飾器(Decorator)

- 一個函數可以有兩個以上的裝飾器

```python
def upperGreeting(func):
    def newFunc(args):
        print(f'This is function "upperGreeting".')
        oldRes = func(args)
        newRes = oldRes.upper()
        print(f"函數名稱：{func.__name__}")
        print(f"函數參數：{args}")
        return newRes
    return newFunc

def bold(func):
    def wrapper(args):
        print(f'This is function "bold".')
        return "bold-" + func(args) + "-bold"
    return wrapper

@upperGreeting
@bold
def greeting(string):
    return string

print(greeting("Hello Python!!!"))
```

執行順序為 upperGreeting -> bold

```
This is function "upperGreeting".
This is function "bold".
函數名稱：wrapper
函數參數：Hello Python!!!
BOLD-HELLO PYTHON!!!-BOLD
```

11-14：動手練習

- 最大公因數（Greatest Common Divisor, GCD)：
  - 輾轉相除法：
    1. 較大的數當作被除數，較小的數當作除數。
    2. 兩數相除
    3. 餘數當作新的除數，原本的除數變成被除數。
    4. 重複 2 跟 3 直到餘數為 0，此時的除數就是最大公因數。
  - 設計一函數 GCD，使用輾轉相除法來計算兩整數的最大公因數。
  - 使用遞迴設計 GCD 函數。

11-14：動手練習

- 河內塔
  - 有三根杆子A，B，C。A杆上有 N 個 (N>1) 穿孔圓盤，盤的尺寸由下到上依次變小。要求按下列規則將所有圓盤移至 C 杆：
    1. 每次只能移動一個圓盤；
    2. 大盤不能疊在小盤上面。
  - 最早發明這個問題的人是法國數學家愛德華·盧卡斯。
    - 傳說越南河內某間寺院有三根銀棒，上串 64 個金盤。寺院裡的僧侶依照一個古老的預言，以上述規則移動這些盤子；預言說當這些盤子移動完畢，世界就會滅亡。這個傳說叫做梵天寺之塔問題（Tower of Brahma puzzle）。
    - 若傳說屬實，僧侶們需要2^{64}-1步才能完成這個任務；若他們每秒可完成一個盤子的移動，就需要 5849 億年才能完成。
    - 這個傳說有若干變體：寺院換成修道院、僧侶換成修士等等。寺院的地點眾說紛紜，其中一說是位於越南的河內，所以被命名為「河內塔」。另外亦有「金盤是創世時所造」、「僧侶們每天移動一盤」之類的背景設定。

11-14：動手練習

- 解法的基本思想是遞迴。假設有 A、B、C 三個塔，A 塔有 *N* 塊盤，目標是把這些盤全部移到 C 塔。那麼先把 A 塔頂部的 *N*-1 塊盤移動到 B 塔，再把 A 塔剩下的大盤移到 C，最後把 B 塔的 *N*-1塊盤移到 C。
- 如此遞迴地使用下去, 就可以求解。

由 André Karwath aka Aka - 自己的作品, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=24669

由 André Karwath aka Aka - 自己的作品, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=85401