1 進階程式

1.1 recursion

In [1]:
```
def fact(x):
    if x == 1: return 1
    else: return x * fact(x-1)

print ('The factorial of 5 is', fact(5))
```

```
The factorial of 5 is 120
```

In [2]:
```
#another good example
def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, m % n)

print(gcd(20, 30)) # 顯示 10
```

```
10
```

In [ ]:
```

```

2 函數的參數宣告方式與呼叫時引數寫法有多種

以下幾種形式：

- 不帶默認值的參數：def func(a): pass
- 帶有默認值的參數：def func(a, b = 1): pass
- 有任意 個數參數*c：def func(a, b = 1, *c): pass

- 有任意 鍵值參數*d : def func(a, b = 1, *c, *d): pass

呼叫(調用)方式:

- 沒有關鍵詞的引數：func("Tom", 20)
- 帶有關鍵詞的引數：(皆帶有關鍵詞可以不考慮順序，這不會搞錯!)
  - func(a = "Tom" , b = 20) 或
  - func(b = 20, a = "Tom")

- 可以混用：func("Tom", b = 20) 先認定位置引述，後面帶關鍵詞的引數可以是任意位置
- 但是-- 位置引數不能在關鍵詞引數之後出現-->這合理，因為這樣寫位置會錯亂了!

  fun(a="Tom", 20) ==> SyntaxError: positional argument follows keyword argument

In [ ]:

```

```

3 函數參數: 個數不一的參數

```
兩種:

(1) *args    參數個數有任意個

(2) ** kwargs  參數可以給予像是字典格式的索引
```

The single asterisk form (*args) is used to pass a non-keyworded, variablelength argument list, and the double asterisk form is used to pass a keyworded, variable-length argument list.

In [ ]:

```

```

In [3]:
```
def var_args(farg, *args):
    print( "formal arg:", farg )
    for arg in args:
        print("another arg:", arg)
var_args(1, "two", 3)
```

```
formal arg: 1
another arg: two
another arg: 3
```

In [ ]:

In [4]:
```
args = ("two", 3)
var_args(1, args)
```

```
formal arg: 1
another arg: ('two', 3)
```

In [ ]:

In [5]:
```
args = ("two", 3) #tuple 若要把引數
var_args(1, *args)
```

```
formal arg: 1
another arg: two
another arg: 3
```

In [6]:
```
args = {"two", 3} #字典dict
var_args(1, *args)
```

```
formal arg: 1
another arg: 3
another arg: two
```

In [7]:
```
args = ["two", 3] #list
var_args(1, *args)
```

```
formal arg: 1
another arg: two
another arg: 3
```

In [ ]:

In [ ]:

In [8]:
```
def var_kwargs(farg, **kwargs):
    print( "formal arg:", farg )
    for key in kwargs:
        print ("another keyword arg: %s: %s" % (key, kwargs[key]))
```

In [9]:
```
# 可以給予key的方式呼叫
var_kwargs(farg=1, myarg2="two", myarg3=3)
```

```
formal arg: 1
another keyword arg: myarg2: two
another keyword arg: myarg3: 3
```

In [ ]:

In [10]:
```
# 用dict格式
kwargs = {"arg3": 3, "arg2": "two"}
var_kwargs(1, **kwargs)
```

```
formal arg: 1
another keyword arg: arg3: 3
another keyword arg: arg2: two
```

In [ ]:

In [ ]:

```

```

In [ ]:

```

```

In [11]:

```
def test_var_args_call(arg1, arg2, arg3):
    print ("arg1:", arg1)
    print ("arg2:", arg2)
    print ("arg3:", arg3)
```

In [12]:

```
kwargs2 = {"arg3": 3, "arg2": "two"}
test_var_args_call(1, **kwargs2)
```

```
arg1: 1
arg2: two
arg3: 3
```

In [ ]:

```

```

In [ ]:

```

```

4 yield 的用法

yield 和 return 很像

return 時，程式主導權回到呼叫該函數的手上，離開就忘記，下次再回來重新開始。 (stack 就會被清除)

yield 會把程式主導權交給呼叫該函數的手上，但下次呼叫時，可以從上次未執行的部分開始執行。離開後沒忘記，下次回來還記得繼續執行下一行。

利用產生器物件(generator)來節省記憶體空間，當資料是依序計算或一批一批讀取資料進入記憶題的話，就是使用產生器的時機。例如:深度學習在訓練模型時，使用generator物件可以節省記憶體。

In [13]:
```
# 以下為使用 return and yield實例，可以看出明顯的不同。
```

In [14]:
```
def fib(n):
    L = []
    i, a, b = 0, 0, 1
    while i < n:
        L.append(b)
        a, b = b, a + b
        i += 1

    return L

print(fib(10))
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

In [15]:
```python
def fib2(n):
    i, a, b = 0, 0, 1
    while True:
        if n <= 0 or i == n:
            break
        a, b = b, a + b
        yield a
        i += 1

d = fib2(10)
for i in d:
    print(i)
```

```
1
1
2
3
5
8
13
21
34
55
```

In [16]:
```python
d = fib2(10)
next(d)
```

Out[16]:
```
1
```

In [17]:
```python
next(d)
```

Out[17]:
```
1
```

In [ ]:
```python

```