**Python陷阱**

#### **1、小括弧—()**

python中，小括弧還能表示元組（tuple）這一資料類型, 元組是immutable的序列。



但如果只有一個元素呢？

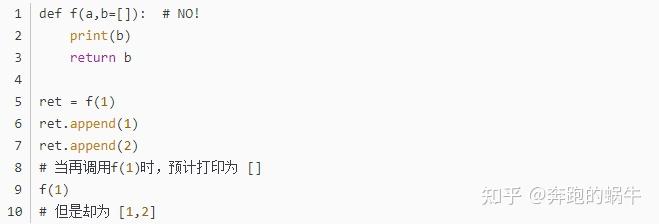


神奇不神奇？如果要表示只有一個元素的元組，正確：



#### **2、默認參數設為空**

含有預設參數的函數，如果類型為容器，且設置為空：



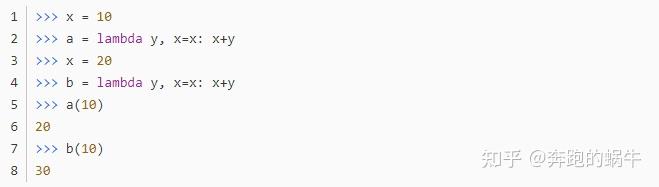
這是可變類型的默認參數之坑，記得設置此類默認參數為None：



#### **3、lambda運算式中使用注意**



在這裡a(10)和b(10)輸出結果相同的原因是：x 在lambda運算式中是一個自由變數，在運行時確定，而不是定義的時候（the value of x used in the lambda expression is a free variable that gets bound at runtime, not definition time），如果需要保存 x 的值，則：



#### **4、列表刪除**

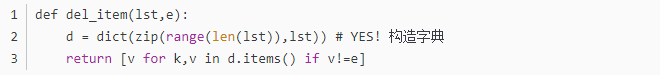
刪除一個清單中的元素，此元素可能在清單中重複多次：



考慮刪除這個序列[1,3,3,3,5]中的元素3，結果發現只刪除其中兩個：



正確：



#### **5、decorator使用注意**

寫一個decorator的時候，最好在實現之前加上functools的wrap，它能保留原有函數的名稱和docstring，例如：



運行輸出：



如果去掉了 @wraps(f) 這一行，得到的結果會是：



#### **6、dict怎麼添加？沒有insert，也沒有append**

list添加一個元素很容易，像下面這樣：



dict怎麼添加？沒有insert，也沒有append，怎麼辦？

直接寫一個K-V組合，就自動添加進來了



別告訴我你不知道怎麼刪除，del

#### **7、共用變數未綁定**

有時想要多個函數共用一個全域變數，但卻在某個函數內試圖修改它為區域變數：



應該在f函數內顯示聲明i為global變數：



#### **8、用isinstance代替type**

type和isinstance均可用於驗證Python對象類型。但是，這是一個顯著的區別：在解決物件類型時，isinstance確保繼承，而type則不保證繼承。

#### **9、可變的默認參數**

在Python中，有兩種類型的物件，在運行時，可變物件可以更改其狀態或內容，而不可變物件則不能。大多數內置對象類型都是不可變的，包括int，float，string，bool和tuple。

執行函式定義後，將檢查默認的Python參數一次，這意味著在定義函數時僅對運算式進行一次檢查，並且對於每個後續調用都使用相同的值。但是，如果更改了可變的預設參數（清單，字典等），則所有後續調用都會更改它。

#### **10、生成多維元素清單**

這個有點像二維陣列，當然生成一個元素是字典的清單也是可以的，更通俗的說，生成一個元素是可變物件的序列

很簡單嘛：

>>> a= [[]] \* 10

>>> a

[[], [], [], [], [], [], [], [], [], []]

>>> a[0].append(10)

>>> a[0]

[10]

看起來很不錯，簡單明瞭，but

>>> a[1]

[10]

>>> a

[[10], [10], [10], [10], [10], [10], [10], [10], [10], [10]]

我猜，這應該不是你預期的結果吧，究其原因，還是因為python中list是可變物件，上述的寫法大家都指向的同一個可變物件，正確的姿勢

>>> a = [[] for \_ in range(10)]

>>> a[0].append(10)

>>> a

[[10], [], [], [], [], [], [], [], [], []]

另外一個在實際編碼中遇到的問題，dict.fromkeys, 也有異曲同工之妙：創建的dict的所有values指向同一個物件。

fromkeys(seq[, value])

Create a new dictionary with keys from seq and values set to value.

#### **11，在訪問列表的時候，修改列表**

列表（list）在python中使用非常廣泛，當然經常會在訪問列表的時候增加或者刪除一些元素。比如，下麵這個函數，試圖刪掉清單中為3的倍數的元素：

>>> def modify\_lst(lst):

... for idx, elem in enumerate(lst):

... if elem % 3 == 0:

... del lst[idx]

...

測試一下，

>>> lst = [1,2,3,4,5,6]

>>> modify\_lst(lst)

>>> lst

[1, 2, 4, 5]

好像沒什麼錯，不過這只是運氣好

>>> lst = [1,2,3,6,5,4]

>>> modify\_lst(lst)

>>> lst

[1, 2, 6, 5, 4]

上面的例子中，6這個元素就沒有被刪除。如果在modify\_lst函數中print idx， item就可以發現端倪：lst在變短，但idx是遞增的，所以在上面出錯的例子中，當3被刪除之後，6變成了lst的第2個元素（從0開始）。在C++中，如果遍歷[容器](https://cloud.tencent.com/product/tke?from_column=20065&from=20065)的時候用反覆運算器刪除元素，也會有同樣的問題。

如果邏輯比較簡單，使用list comprehension是不錯的注意

#### **12，閉包與lambda**

這個也是老生長談的例子，在其他語言也有類似的情況。先看一個例子:

>>> def create\_multipliers():

... return [lambda x:i\*x for i in range(5)]

...

>>> for multiplier in create\_multipliers():

... print multiplier(2)

...

create\_multipliers函數的返回值時一個列表，清單的每一個元素都是一個函數 －－ 將輸入參數x乘以一個倍數i的函數。預期的結果時0，2，4，6，8. 但結果是5個8，意外不意外。

由於出現這個陷阱的時候經常使用了lambda，所以可能會認為是lambda的問題，但lambda表示不願意背這個鍋。問題的本質在與python中的屬性查找規則，LEGB（local，enclousing，global，bulitin），在上面的例子中，i就是在閉包作用域（enclousing），而Python的閉包是 遲綁定 ， 這意味著閉包中用到的變數的值，是在內建函式被調用時查詢得到的。

解決辦法也很簡單，那就是變閉包作用域為局部作用域。

>>> def create\_multipliers():

... return [lambda x, i = i:i\*x for i in range(5)]

...

#### **13，定義del**

大多數電腦專業的同學可能都是先學的C、C++，構造、析構函數的概念應該都非常熟。於是，當切換到python的時候，自然也想知道有沒有相應的函數。比如，在C++中非常有名的RAII，即通過構造、析構來管理資源（如記憶體、檔描述符）的聲明週期。那在python中要達到同樣的效果怎麼做呢，即需要找到一個物件在銷毀的時候一定會調用的函數，於是發現了**init**, **del**函數，可能簡單寫了兩個例子發現確實也能工作。但事實上可能掉進了一個陷阱，在python documnet是有描述的：

Circular references which are garbage are detected when the option cycle detector is enabled (it’s on by default), but can only be cleaned up if there are no Python-level **del**() methods involved.

簡單來說，如果在迴圈引用中的物件定義了**del**,那麼python gc不能進行回收，因此，存在記憶體洩漏的風險

#### **14，不同的姿勢import同一個module**

示例在stackoverflow的例子上稍作修改，假設現在有一個package叫mypackage，裡面包含三個python檔：mymodule.py, main.py, **init**.py。mymodule.py代碼如下：

l = []

class A(object):

pass

main.py代碼如下：

def add(x):

from mypackage import mymodule

mymodule.l.append(x)

print "updated list",mymodule.l, id(mymodule)

def get():

import mymodule

print module in get , id(mymodule)

return mymodule.l

if \_\_name\_\_ == \_\_main\_\_ :

import sys

sys.path.append( ../ )

add(1)

ret = get()

print "lets check", ret

運行python main.py，結果如下：

updated list [1] 4406700752

module in get 4406700920

lets check []

從運行結果可以看到，在add 和 get函數中import的mymodule不是同一個module，ID不同。當然，在python2.7.10中，需要main.py的第13行才能出現這樣的效果。你可能會問，誰會寫出第13行這樣的代碼呢？事實上，在很多專案中，為了import的時候方便，會往sys.path加入一堆路徑。那麼在專案中，大家同意一種import方式就非常有必要了

#### **15，python升級**

python3.x並不向後相容，所以如果從2.x升級到3.x的時候得小心了，下面列舉兩點：

在python2.7中，range的返回值是一個列表；而在python3.x中，返回的是一個range物件。

map()、filter()、 dict.items()在python2.7返回列表，而在3.x中返回反覆運算器。當然反覆運算器大多數都是比較好的選擇，更加pythonic，但是也有缺點，就是只能遍歷一次。在instagram的分享中，也提到因為這個導致的一個坑爹的bug。

#### **16：++i —i**

這個陷阱主要是坑來自C、C++背景的同學。簡單來說，++i是對i取兩次正號，—i是對i取兩次負號，運算完之後i的值不變。

#### **17：setattr** **getattr** **getattribute**

Python中有大量的magic method（形似**xx**），其中許多跟屬性訪問有關，比如**get**， **set**，**delete\_，**getattr**,** setattr**,** delattr**,** getattribute**。前三個跟descriptor相關，詳細可參見《python descriptor 詳解》。坑爹的是，**getattr**與**setattr**相差很大，在《python屬性查找（attribute look up）》一文中有詳細介紹。簡單說來，**setattr**與**getattribute**是對應的，都是修改python預設的屬性修改、查找機制，而**getattr**只是預設查找機制無法找到屬性的時候才會調用，**setattr**應該叫**setattribute\_\_才恰當！

#### **18：元素列表生成**

>>> buf= []\*5

結果:

...buf

[]

正確

>>> buf= [“”]\*5

結果:

...buf

[“”, “”, “”, “”, “”]

#### **19：元素列表複製**

>>> buf= [[y for x in range(2)] for y in range(5)]

buf1=buf

結果:

buf [[0, 0], [1, 1], [2, 2], [3, 3], [4, 4]]

buf1 [[0, 0], [1, 1], [2, 2], [3, 3], [4, 4]]

但是

>>>id (buf)

…2339005404736

>>>id (buf1)

…2339005404736

位址相同,不是需要結果

正確

>>> import copy

buf1=copy.deepcopy(buf)

結果:

>>>id (buf)

…2339005404736

>>>id (buf1)

…2339003865344