# Python技巧

**iterables**

根據Python標準文檔，Iterable的概念如下：

一種能夠一次返回一個成員的物件。

iterables的示例包括：

* 所有序列類型（如list、str和tuple）
* 一些非序列類型，如dict、檔物件以及類的實現中定義了\_\_iter\_\_()方法

Iterables是一個需要我們牢記的概念，因為接下來我們展示的許多技巧都使用itertools包。

itertools模組提供了一些函數，用於接收Iterable物件，而不僅僅是列印逐個對象。

## 1.

在工作學習中，我們經常會需要使用一個簡單的函數來實現從一個list來生成新的list, set或dict.此時我們就會用到iterables概念。

舉例來說：

生成List:

names = ['John', 'Bard', 'Jessica' 'Andres']

lower\_names = [name.lower() for name in names]

生成Set:

names = ['John', 'Bard', 'Jessica' 'Andres']

lower\_names = {

name.lower() for name in names}

生成Dict:

names = ['John', 'Bard', 'Jessica' 'Andres']

lower\_names = {

name:name.lower() for name in names}

個人建議：

僅當for語句、函式呼叫和方法調用的數量較少時使用。

## 2.

有時，我們需要獲得兩個清單物件之間的所有可能組合。

我們首先想到的實現可能如下：

l1 = [1, 2, 3]

l2 = [4, 5, 6]

combinations = []

for e1 in l1:

for e2 in l2:

combinations.append((e1, e2))

或者簡化一下，如下：

combinations = [(e1, e2) for e1 in l1 for e2 in l1]

上述實現已經很簡潔了，但標準庫itertools提供product函數，從而提供了相同的結果。如下所示：

from itertools import product

l1 = [1, 2, 3]

l2 = [4, 5, 6]

combinatios = product(l1, l2)

## 3.

假設有一個元素清單，我們需要在每對相鄰元素之間比較或應用一些操作，這有時稱為2個元素的滑動視窗。我們可以採用以下方式：

from itertools import tee

from typing import Iterable

def window2(iterable: Iterable):

it, offset = tee(iter(iterable))

next(offset)

return zip(it, offset)

l = [1, 2, 3, 4, 5, 6]

dd = window2(l)

for a in dd:

print(a)

運行結果如下：

(1, 2)

(2, 3)

(3, 4)

(4, 5)

(5, 6)

## 4.

有時，我們會需要一個類來存儲資訊，但是如果我們覺得創建一個類並定義其\_\_init\_\_()函數太麻煩時，我們不妨選擇使用dataclass。如下所示：

from dataclasses import dataclass

@dataclass

class Person:

name: str

age: int

address: str

上述代碼創建了一個具有預設構造函數的類，該類以與聲明相同的順序接收相應欄位的賦值。

person = Person(name='John', age=12, address='nanjing street')

dataclass的另一個優點是，預設情況下，會生成特殊方法，如\_\_str\_\_、 **repr** 、\_\_eq\_\_等。關於dataclass的更多用法，可以 ***[參考官網](https://link.zhihu.com/?target=https%3A//docs.python.org/3/library/dataclasses.html" \t "_blank)*** 。

值得一提的是我們在類中聲明的成員變數的類型注釋（str、int等）並不強制在構造函數中傳遞的值屬於這種類型。也就是說dataclasses構造物件時並不執行資料類型的檢查。

## 5.

我們有時希望將一個物件上的操作視為tuple上的操作，一種選擇是使用collections.namedtuple,但也存在更類似於dataclass的實現。如下：

from typing import NamedTuple

class Coordinate(NamedTuple):

x: int

y: int

上述定義了一個標準的類可以被當做tuple來使用，如下：

coordinate = Coordinate(10, 15)

coordinate.x == coordinate[0] // True

coordinate.y == coordinate[1] // True

## 6.

假如我們有一個dataclass，需要驗證輸入資料是否符合類型注釋。在這種情況下，安裝協力廠商套裝軟體pydantic並將

*from dataclasses import dataclass* 替換為 *from pydantic.dataclasses import dataclass* 即可，如下：

from pydantic.dataclasses import dataclass

@dataclass

class Person:

name: str

age: int

address: str

這將生成一個類，該類具有根據成員變數聲明的類型進行輸入資料的解析和類型驗證。 *Pydantic* 在運行時強制執行類型提示，並在資料無效時提供友好的錯誤提醒。

## 7.

在某些情況下，我們需要生成一些容器中元素頻率的基本統計資訊。在這種情況下，您可以使用標準結構Counter來接收iterable並根據元素的頻率生成相應的統計資訊。

from collections import Counter

l = [1, 1, 2, 3, 4, 4]

frequencys = Counter(l)

print(frequencys[1]) // Ouput: 2

print(frequencys[2]) // Ouput: 1

print(frequencys[2323]) // Ouput: 0

Counter也提供了一些其他方法，比如如most\_common，用於檢索最常見的元素。

## 8.

如果我們相對兩個list中的元素對做相應的函數處理，我們最容易想到的方法如下：

l1 = [1, 2, 3]

l2 = [4, 5, 6]

for (e1, e2) in zip(l1, l2):

f(e1, e2)

但是使用函數map可以讓代碼更加簡潔一些。

l1 = [1, 2, 3]

l2 = [4, 5, 6]

map(f, l1, l2)

## 9.

有時候我們需要從一個list中隨機選擇一個元素，此時我們使用random.choice.如下所示：

from random import choice

l = [1, 2, 3]

random = choice(l)

如果我們需要隨機選擇多個元素呢？當然是使用random.choices.

from random import choices

l = [1, 2, 3, 4, 5]

random\_elements = choices(l, k=3)

上述代碼中的參數k為我們隨機選擇元素的個數。

* 字串操作
* 列表推導式
* lambda運算式
* 映射函數
* 單行實現if、elif和else條件判斷
* 拉鍊函數

## 1、字串操作

可以使用算數運算式來操作字串，例如 + 和 \* ：

>>> my\_string = "Hi Medium..!"

>>> print(my\_string \* 2)

Hi Medium..!Hi Medium..!

>>> print(my\_string + " I love Python" \* 2)

Hi Medium..! I love Python I love Python

也可以使用[::-1]來輕鬆地翻轉字串，而且不局限於字串！

>>> print(my\_string[::-1])

!..muideM iH>>> my\_list = [1,2,3,4,5]

>>> print(my\_list[::-1])

[5, 4, 3, 2, 1]

翻轉操作可以應用於單字清單嗎？當然！我們可以開發一個Yoda翻譯器：

>>> word\_list = ["awesome", "is", "this"]

>>> print(' '.join(word\_list[::-1]) + '!')

this is awesome!

上面我們使用.join()方法用空白字元號反向連接列表中的所有成員，最後再添加一個感嘆號。

## 2、列表推導式

你知道嗎，當我掌握這個技巧時，我感覺整個世界都變了！這種操作清單 的方式真的太強大、直觀而且可讀性高。

例如，假設我們有一個函數來計算輸入參數的平方再加5：

>>> def stupid\_func(x):

>>> return x\*\*2 + 5

現在我們要用這個函數處理一個清單中的所有奇數序號的成員，如果你不知道 列表推導式，可能會這樣來實現：

>>> my\_list = [1, 2, 3, 4, 5]

>>> new\_list = []

>>> for x in my\_list:

>>> if x % 2 != 0:

>>> new\_list.append(stupid\_func(x))

>>> print(new\_list)

[6, 14, 30]

但是有更簡單的實現方法！

>>> my\_list = [1, 2, 3, 4, 5]

>>> print([stupid\_func(x) for x in my\_list if x % 2 != 0])

[6, 14, 30]

清單推導式的語法是

[ expression for item in list ]

還可以使用額外的條件判斷運算式：

[ expression for item in list if conditional ]

上面這個運算式實際上等價於：

>>> for item in list:

>>> if conditional:

>>> expression

實際上你還可以進一步優化，因為我們其實不需要那個stupid\_func：

>>> print([x \*\* 2 + 5 for x in my\_list if x % 2 != 0])

[6, 14, 30]

## 3、Lambda運算式

Lambda看起來有點怪異，不過和這篇文章中的其他內容一樣，Lambda 超級強大，而且一旦你理解了就會發現它其實很直觀。

基本上Lambda運算式就是一個匿名函數。為什麼要匿名？這是因為Lambda 運算式通常應用於執行簡單的操作，因此使用def來正式定義一個函數 就顯得有點累贅了。

讓我們以上面的計算為例看一下Lambda的使用方法。在上面的代碼中 我們使用正式的函式宣告語法def stupid\_func(x)，現在讓我們使用 Lambda運算式：

>>> stupid\_func = (lambda x : x \*\* 2 + 5)

>>> print([stupid\_func(1), stupid\_func(3), stupid\_func(5)])

[6, 14, 30]

為什麼要使用這麼怪異的語法？Lambda的好處在於你可以執行一些簡單 的操作而無需提前定義一個函數。例如，假設有一組數值，我們如何 用Python來排序？一種方法是使用sorted()函數：

>>> my\_list = [2, 1, 0, -1, -2]

>>> print(sorted(my\_list))

[-2, -1, 0, 1, 2]

sorted()默認是按昇冪對列表成員排序，但是如果我們希望按成員的 平方和排序，就可以使用lambda運算式了：

>>> print(sorted(my\_list, key = lambda x : x \*\* 2))

[0, -1, 1, -2, 2]

## 4、映射函數 map

映射函數map()可以使用指定的函數來處理像清單之類的成員序列。例如， 假設我們希望得到兩個列表中對應成員的乘積的結果列表，應該怎麼做？ 使用Lambda運算式和映射函數就可以了！

>>> print(list(map(lambda x, y : x \* y, [1, 2, 3], [4, 5, 6])))

[4, 10, 18]

如果不使用Lambda和map的話，代碼就沒這麼優雅了：

>>> x, y = [1, 2, 3], [4, 5, 6]

>>> z = []

>>> for i in range(len(x)):

>>> z.append(x[i] \* y[i])

>>> print(z)

[4, 10, 18]

## 5、單行條件判斷

在你的代碼中可能會有這樣的條件判斷邏輯：

>>> x = int(input())

>>> if x >= 10:

>>> print("Horse")

>>> elif 1 < x < 10:

>>> print("Duck")

>>> else:

>>> print("Baguette")

運行上面的代碼，會提示我們輸入，假設輸入5的話，就會顯示 Duck。但是我們可以只用一行代碼就實現同樣的功能！

print("Horse" if x >= 10 else "Duck" if 1 < x < 10 else "Baguette")

實在是妙！

## 6、拉鍊函數 zip

上面的映射函數map()對兩個列表中並行地執行某個指定的處理函數。 拉鍊函數zip()比這還要簡單。

假設我們有兩個清單，一個記錄人名，另一個記錄姓氏，那麼我們應該 如何合併得到一個全名列表？答案是使用zip()！

>>> first\_names = ["Peter", "Christian", "Klaus"]

>>> last\_names = ["Nistrup", "Smith", "Jensen"]

>>> print([' '.join(x) for x in zip(first\_names, last\_names)])

['Peter Nistrup', 'Christian Smith', 'Klaus Jensen']

**1、for迴圈中的else條件**

這是一個for-else方法，迴圈遍歷列表時使用else語句。

下面舉個例子，比如我們想檢查一個列表中是否包含奇數。

那麼可以通過for迴圈，遍歷查找。

numbers = [2, 4, 6, 8, 1]

for number in numbers:

if number % 2 == 1:

print(number)

break

else:

print("No odd numbers")

如果找到了奇數，就會列印該數值，並且執行break語句，跳過else語句。

沒有的話，就不會執行break語句，而是執行else語句。

**▍2、從清單中獲取元素，定義多個變數**

my\_list = [1, 2, 3, 4, 5]

one, two, three, four, five = my\_list

**▍3、使用heapq模組，獲取清單中n個最大或最小的元素**

import heapq

scores = [51, 33, 64, 87, 91, 75, 15, 49, 33, 82]

print(heapq.nlargest(3, scores)) # [91, 87, 82]

print(heapq.nsmallest(5, scores)) # [15, 33, 33, 49, 51]

**▍4、將清單中的所有元素作為參數傳遞給函數**

我們可以使用 \* 號，提取清單中所有的元素

my\_list = [1, 2, 3, 4]

print(my\_list) # [1, 2, 3, 4]

print(\*my\_list) # 1 2 3 4

如此便可以將清單中的所有元素，作為參數傳遞給函數

def sum\_of\_elements(\*arg):

total = 0

for i in arg:

total += i

return total

result = sum\_of\_elements(\*[1, 2, 3, 4])

print(result) # 10

**▍5、獲取清單的所有中間元素**

\_, \*elements\_in\_the\_middle, \_ = [1, 2, 3, 4, 5, 6, 7, 8]

print(elements\_in\_the\_middle) # [2, 3, 4, 5, 6, 7]

**▍6、使用一行代碼賦值多個變數**

one, two, three, four = 1, 2, 3, 4

**▍7、列表推導式**

只用一行代碼，便可完成對陣列的反覆運算以及運算。

比如，將清單中的每個數字提高一倍。

numbers = [1, 2, 3, 4, 5]

squared\_numbers = [num \* num for num in numbers]

print(squared\_numbers) # [1, 4, 9, 16, 25]

推導式不僅清單能用，字典、集合、生成器也能使用。

下面看一下，使用字典推導式，將字典的值提高一倍。

dictionary = {'a': 4, 'b': 5}

squared\_dictionary = {key: num \* num for (key, num) in dictionary.items()}

print(squared\_dictionary) # {'a': 16, 'b': 25}

**▍8、通過Enum枚舉同一標籤或一系列常量的集合**

枚舉是綁定到唯一的常量值的一組符號名稱(成員)。

在枚舉中，成員可以通過身份進行比較，枚舉本身可以反覆運算。

from enum import Enum

class Status(Enum):

NO\_STATUS = -1

NOT\_STARTED = 0

IN\_PROGRESS = 1

COMPLETED = 2

print(Status.IN\_PROGRESS.name) # IN\_PROGRESS

print(Status.COMPLETED.value) # 2

**▍9、重複字串**

name = "Banana"

print(name \* 4) # BananaBananaBananaBanana

**▍10、比較3個數字的大小**

如果想比較一個值和其他兩個值的大小情況，你可以使用簡單的數學運算式。

1 < x < 10

這個是最簡單的代數運算式，在Python中也是可以使用的。

x = 3

print(1 < x < 10) # True

print(1 < x and x < 10) # True

**▍11、使用1行代碼合併字典**

first\_dictionary = {'name': 'Fan', 'location': 'Guangzhou'}

second\_dictionary = {'name': 'Fan', 'surname': 'Xiao', 'location': 'Guangdong, Guangzhou'}

result = first\_dictionary | second\_dictionary

print(result)

# {'name': 'Fan', 'location': 'Guangdong, Guangzhou', 'surname': 'Xiao'}

**▍12、查找元組中元素的索引**

books = ('Atomic habits', 'Ego is the enemy', 'Outliers', 'Mastery')

print(books.index('Mastery')) # 3

**▍13、將字串轉換為字串清單**

假設你在函數中獲得輸出，原本應該是一個列表，但實際上卻是一個字串。

input = "[1,2,3]"

你可能第一時間會想到使用索引或者規則運算式。

實際上，使用ast模組的literal\_eval方法就能搞定。

import ast

def string\_to\_list(string):

return ast.literal\_eval(string)

string = "[1, 2, 3]"

my\_list = string\_to\_list(string)

print(my\_list) # [1, 2, 3]

string = "[[1, 2, 3],[4, 5, 6]]"

my\_list = string\_to\_list(string)

print(my\_list) # [[1, 2, 3], [4, 5, 6]]

**▍14、計算兩數差值**

計算出2個數字之間的差值。

def subtract(a, b):

return a - b

print((subtract(1, 3))) # -2

print((subtract(3, 1))) # 2

上面的這個方法，需要考慮數值的先後順序。

def subtract(a, b):

return a - b

print((subtract(a=1, b=3))) # -2

print((subtract(b=3, a=1))) # -2

使用具名引數，安排順序，這樣就不會出錯了。

**▍15、用一個print()語句列印多個元素**

print(1, 2, 3, "a", "z", "this is here", "here is something else")

**▍16、在同一行列印多個元素**

print("Hello", end="")

print("World") # HelloWorld

print("Hello", end=" ")

print("World") # Hello World

print('words', 'with', 'commas', 'in', 'between', sep=', ')

# words, with, commas, in, between

**▍17、列印多個值，在每個值之間使用自訂分隔符號**

print("29", "01", "2022", sep="/") # 29/01/2022

print("name", "domain.com", sep="@") # name@domain.com

**▍18、不能在變數名的開頭使用數位**

four\_letters = "abcd" # this works

4\_letters = "abcd" # this doesn’t work

這是Python的變數命名規則。

**▍19、不能在變數名的開頭使用運算子**

+variable = "abcd" # this doesn’t work

**▍20、數字的第一位不能是0**

number = 0110 # this doesn't work

這個確實挺神奇的。

**▍21、在變數名的任何地方使用底線**

a\_\_\_\_\_\_b = "abcd" # this works

\_a\_b\_c\_d = "abcd" # this also works

這並不意味著，你可以無限使用，為了代碼的易讀性，還是需要合理使用。

**▍22、使用底線分割數值較大的數位**

print(1\_000\_000\_000) # 1000000000

print(1\_234\_567) # 1234567

如此，看到一大堆數字時，也能輕鬆閱讀。

**▍23、反轉列表**

my\_list = ['a', 'b', 'c', 'd']

my\_list.reverse()

print(my\_list) # ['d', 'c', 'b', 'a']

**▍24、使用步進函數對字串切片**

my\_string = "This is just a sentence"

print(my\_string[0:5]) # This

# Take three steps forward

print(my\_string[0:10:3]) # Tsse

**▍25、反向切片**

my\_string = "This is just a sentence"

print(my\_string[10:0:-1]) # suj si sih

# Take two steps forward

print(my\_string[10:0:-2]) # sjs i

**▍26、使用開始或結束索引進行切片**

my\_string = "This is just a sentence"

print(my\_string[4:]) # is just a sentence

print(my\_string[:3]) # Thi

**▍27、/和//的區別**

print(3/2) # 1.5

print(3//2) # 1

**▍28、==和is的區別**

is：檢查兩個變數是否指向同一物件記憶體中

==：比較兩個物件的值

first\_list = [1, 2, 3]

second\_list = [1, 2, 3]

# 比較兩個值

print(first\_list == second\_list) # True

# 是否指向同一記憶體

print(first\_list is second\_list)

# False

third\_list = first\_list

print(third\_list is first\_list)

# True

**▍29、合併字典**

dictionary\_one = {"a": 1, "b": 2}

dictionary\_two = {"c": 3, "d": 4}

merged = {\*\*dictionary\_one, \*\*dictionary\_two}

print(merged) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

**▍30、檢查字串是否大於另一字串**

first = "abc"

second = "def"

print(first < second) # True

second = "ab"

print(first < second) # False

**▍31、檢查字串是否以特定字元開頭(不使用索引)**

my\_string = "abcdef"

print(my\_string.startswith("b")) # False

**▍32、使用id()查找變數的唯一id**

print(id(1)) # 4325776624

print(id(2)) # 4325776656

print(id("string")) # 4327978288

**▍33、整數、浮點數、字串、布林值和元組都是不可變的**

當變數被賦值為整數、浮點數、字串、布林值、元組這些不可變類型後，該變數就會指向一個記憶體物件。

如果重新給變數再賦值，它的記憶體物件就會發生改變。

number = 1

print(id(number)) # 4325215472

print(id(1)) # 4325215472

number = 3

print(id(number)) # 4325215536

print(id(1)) # 4325215472

**▍34、字串和元組也是不可變的**

此處再說明一次。

name = "Fatos"

print(id(name)) # 4422282544

name = "fatos"

print(id(name)) # 4422346608

**▍35、清單、集合和字典都是可變的**

這意味著發生更改時，不會改變其記憶體物件。

cities = ["Beijing", "Guangzhou", "chengdu"]

print(id(cities)) # 4482699712

cities.append("Beijing")

print(id(cities)) # 4482699712

下麵是字典。

my\_set = {1, 2, 3, 4}

print(id(my\_set)) # 4352726176

my\_set.add(5)

print(id(my\_set)) # 4352726176

**▍36、把一個列表變成不可變的列表**

my\_set = frozenset(['a', 'b', 'c', 'd'])

my\_set.add("a")

使用frozenset()後，你就無法更改了。

**▍37、if-elif塊可以在沒有else塊的情況下存在**

但是elif不能在沒有if語句之前獨立存在。

def check\_number(number):

if number > 0:

return "Positive"

elif number == 0:

return "Zero"

return "Negative"

print(check\_number(1)) # Positive

**▍38、使用sorted()檢查2個字串是否為相同**

def check\_if\_anagram(first\_word, second\_word):

first\_word = first\_word.lower()

second\_word = second\_word.lower()

return sorted(first\_word) == sorted(second\_word)

print(check\_if\_anagram("testinG", "Testing")) # True

print(check\_if\_anagram("Here", "Rehe")) # True

print(check\_if\_anagram("Know", "Now")) # False

**▍39、獲取字元的Unicode值**

print(ord("A")) # 65

print(ord("B")) # 66

print(ord("C")) # 66

print(ord("a")) # 97

**▍40、獲取字典的鍵**

dictionary = {"a": 1, "b": 2, "c": 3}

keys = dictionary.keys()

print(list(keys)) # ['a', 'b', 'c']

**▍41、獲取字典的值**

dictionary = {"a": 1, "b": 2, "c": 3}

values = dictionary.values()

print(list(values)) # [1, 2, 3]

**▍42、交換字典的鍵、值位置**

dictionary = {"a": 1, "b": 2, "c": 3}

reversed\_dictionary = {j: i for i, j in dictionary.items()}

print(reversed) # {1: 'a', 2: 'b', 3: 'c'}

**▍43、將布林值轉換為數位**

print(int(False)) # 0

print(float(True)) # 1.0

**▍44、在算數運算中使用布林值**

x = 10

y = 12

result = (x - False)/(y \* True)

print(result) # 0.8333333333333334

**▍45、將任何資料類型轉換為布林值**

print(bool(.0)) # False

print(bool(3)) # True

print(bool("-")) # True

print(bool("string")) # True

print(bool(" ")) # True

**▍46、將值轉換為複數**

print(complex(10, 2)) # (10+2j)

也可以將數位轉換為十六進位數。

print(hex(11)) # 0xb

**▍47、在列表的第一個位置添加一個值**

如果使用append()，將從列表的最後一個位置插入新值。

可以通過使用insert()，來指定插入新元素的索引和數值。

那麼列表的第一個位置為0，即下標為0。

my\_list = [3, 4, 5]

my\_list.append(6)

my\_list.insert(0, 2)

print(my\_list) # [2, 3, 4, 5, 6]

**▍48、Lambda函數只能在一行代碼中**

無法通過多行代碼，來使用lambda函數。

comparison = lambda x: if x > 3:

print("x > 3")

else:

print("x is not greater than 3")

報錯。

**▍49、Lambda中的條件陳述式應始終包含else語句**

comparison = lambda x: "x > 3" if x > 3

運行上面的代碼，報錯。

這是由於條件運算式的特性，而不是lambda的導致的。

**▍50、使用filter()，獲得一個新物件**

my\_list = [1, 2, 3, 4]

odd = filter(lambda x: x % 2 == 1, my\_list)

print(list(odd)) # [1, 3]

print(my\_list) # [1, 2, 3, 4]

**▍51、map()返回一個新物件**

map()函數將給定函數應用於可反覆運算物件(清單、元組等)，然後返回結果(map對象)。

my\_list = [1, 2, 3, 4]

squared = map(lambda x: x \*\* 2, my\_list)

print(list(squared)) # [1, 4, 9, 16]

print(my\_list) # [1, 2, 3, 4]

**▍52、range()的step參數**

for number in range(1, 10, 3):

print(number, end=" ")

# 1 4 7

**▍53、range()默認從0開始**

def range\_with\_zero(number):

for i in range(0, number):

print(i, end=' ')

def range\_with\_no\_zero(number):

for i in range(number):

print(i, end=' ')

range\_with\_zero(3) # 0 1 2

range\_with\_no\_zero(3) # 0 1 2

**▍54、不需要和0比較長度**

如果長度大於0，則默認為True。

def get\_element\_with\_comparison(my\_list):

if len(my\_list) > 0:

return my\_list[0]

def get\_first\_element(my\_list):

if len(my\_list):

return my\_list[0]

elements = [1, 2, 3, 4]

first\_result = get\_element\_with\_comparison(elements)

second\_result = get\_element\_with\_comparison(elements)

print(first\_result == second\_result) # True

**▍55、可以在同一個作用域內多次定義一個方法**

但是，只有最後一個會被調用，覆蓋以前。

def get\_address():

return "First address"

def get\_address():

return "Second address"

def get\_address():

return "Third address"

print(get\_address()) # Third address

**▍56、在外部直接訪問私有屬性**

在定義屬性或方法時，在屬性名或者方法名前增加兩個底線，定義的就是私有屬性或方法。

如果想要在外部訪問，那麼只需要在名稱前面加上 '\_類名' 變成 '\_類名\_\_名稱'。

class Engineer:

def \_\_init\_\_(self, name):

self.name = name

self.\_\_starting\_salary = 62000

dain = Engineer('Dain')

print(dain.\_Engineer\_\_starting\_salary) # 62000

**▍57、檢查物件的記憶體使用情況**

import sys

print(sys.getsizeof("bitcoin")) # 56

**▍58、定義一個方法，可以調用任意個參數**

def get\_sum(\*arguments):

result = 0

for i in arguments:

result += i

return result

print(get\_sum(1, 2, 3)) # 6

print(get\_sum(1, 2, 3, 4, 5)) # 15

print(get\_sum(1, 2, 3, 4, 5, 6, 7)) # 28

**▍59、使用super()或父類的名稱調用父類的初始化**

使用super函式呼叫父類的初始化方法。

class Parent:

def \_\_init\_\_(self, city, address):

self.city = city

self.address = address

class Child(Parent):

def \_\_init\_\_(self, city, address, university):

super().\_\_init\_\_(city, address)

self.university = university

child = Child('Peking University', 'Fudan University', 'Tsinghua University')

print(child.university) # Tsinghua University

使用父類的名稱調用父類。

class Parent:

def \_\_init\_\_(self, city, address):

self.city = city

self.address = address

class Child(Parent):

def \_\_init\_\_(self, city, address, university):

Parent.\_\_init\_\_(self, city, address)

self.university = university

child = Child('Peking University', 'Fudan University', 'Tsinghua University')

print(child.university) # Tsinghua University

**▍60、在類中使用 + 操作符**

在兩個int資料類型之間使用 + 運算子時，將得到它們的和。

而在兩個字串資料型別之間使用它時，會將其合併。

print(10 + 1) # 兩數相加

print('first' + 'second') # 字串相加

這個就是操作符重載，你還可以在類中使用(\_\_add\_\_)。

class Expenses:

def \_\_init\_\_(self, rent, groceries):

self.rent = rent

self.groceries = groceries

def \_\_add\_\_(self, other):

return Expenses(self.rent + other.rent,

self.groceries + other.groceries)

april\_expenses = Expenses(1000, 200)

may\_expenses = Expenses(1000, 300)

total\_expenses = april\_expenses + may\_expenses

print(total\_expenses.rent) # 2000

print(total\_expenses.groceries) # 500

**▍61、在類中使用 < 和 == 操作符**

下面定義一個操作重載示例( < 操作符)，使用\_\_lt\_\_方法。

class Game:

def \_\_init\_\_(self, score):

self.score = score

def \_\_lt\_\_(self, other):

return self.score < other.score

first = Game(1)

second = Game(2)

print(first < second) # True

同樣的，== 操作符使用\_\_eq\_\_方法。

class Journey:

def \_\_init\_\_(self, location, destination, duration):

self.location = location

self.destination = destination

self.duration = duration

def \_\_eq\_\_(self, other):

return ((self.location == other.location) and

(self.destination == other.destination) and

(self.duration == other.duration))

first = Journey('Location A', 'Destination A', '30min')

second = Journey('Location B', 'Destination B', '30min')

print(first == second)

還有一些其他的定義。

\_\_sub\_\_() for -

\_\_mul\_\_() for \*

\_\_truediv\_\_() for /

\_\_ne\_\_() for !=

\_\_ge\_\_() for >=

\_\_gt\_\_() for >

**▍62、為類的物件定義自訂的可列印版本**

class Rectangle:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def \_\_repr\_\_(self):

return repr('Rectangle with area=' + str(self.a \* self.b))

print(Rectangle(3, 4)) # 'Rectangle with area=12'

**▍63、交換字串中字元的大小寫**

string = "This is just a sentence."

result = string.swapcase()

print(result) # tHIS IS JUST A SENTENCE.

**▍64、檢查字串是否都是空格**

string = " "

result = string.isspace()

print(result) # True

**▍65、檢查字串是否都是字母或數位**

name = "Password"

print(name.isalnum()) # True

name = "Secure Password "

print(name.isalnum()) # False

name = "S3cur3P4ssw0rd"

print(name.isalnum()) # True

name = "133"

print(name.isalnum()) # True

**▍66、檢查字串是否都是字母**

string = "Name"

print(string.isalpha()) # True

string = "Firstname Lastname"

print(string.isalpha()) # False

string = "P4ssw0rd"

print(string.isalpha()) # False

**▍67、根據參數刪除字元**

從右側開始。

string = "This is a sentence with "

print(string.rstrip()) # "This is a sentence with"

string = "this here is a sentence…..,,,,aaaaasd"

print(string.rstrip(".,dsa")) # "this here is a sentence"

同樣的，左側也能操作。

string = "ffffffffFirst"

print(string.lstrip("f")) # First

**▍68、檢查字串是否為數位**

string = "seven"

print(string.isdigit()) # False

string = "1337"

print(string.isdigit()) # True

string = "5a"

print(string.isdigit()) # False

string = "2\*\*5"

print(string.isdigit()) # False

**▍69、檢查字串是否為中文數位**

# 42673

string = "四二六七三"

print(string.isdigit()) # False

print(string.isnumeric()) # True

**▍70、檢查字串是否所有單詞都是大寫開頭**

string = "This is a sentence"

print(string.istitle()) # False

string = "10 Python Tips"

print(string.istitle()) # True

string = "How to Print A String in Python"

# False

print(string.istitle())

string = "PYTHON"

print(string.istitle()) # False

**▍71、在元組中使用負索引**

numbers = (1, 2, 3, 4)

print(numbers[-1]) # 4

print(numbers[-4]) # 1

**▍72、在元組中嵌套列表和元組**

mixed\_tuple = (("a"\*10, 3, 4), ['first', 'second', 'third'])

print(mixed\_tuple[1]) # ['first', 'second', 'third']

print(mixed\_tuple[0]) # ('aaaaaaaaaa', 3, 4)

**▍73、快速統計元素在清單中出現的次數**

names = ["Besim", "Albert", "Besim", "Fisnik", "Meriton"]

print(names.count("Besim")) # 2

**▍74、使用slice()獲取元素**

使用slice()獲取最後n個元素。

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

slicing = slice(-4, None)

print(my\_list[slicing]) # [4, 5, 6]

print(my\_list[-3]) # 4

使用slice()做切片任務。

string = "Data Science"

slice\_object = slice(5, None)

print(string[slice\_object]) # Science

**▍75、計算元素在元組中出現的次數**

my\_tuple = ('a', 1, 'f', 'a', 5, 'a')

print(my\_tuple.count('a')) # 3

**▍76、獲取元組中元素的索引**

my\_tuple = ('a', 1, 'f', 'a', 5, 'a')

print(my\_tuple.index('f')) # 2

**▍77、步進獲得元組**

my\_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

print(my\_tuple[::3]) # (1, 4, 7, 10)

**▍78、通過索引獲取子元組**

my\_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

print(my\_tuple[3:]) # (4, 5, 6, 7, 8, 9, 10)

**▍79、將清單、集合、字典中所有元素刪除**

my\_list = [1, 2, 3, 4]

my\_list.clear()

print(my\_list) # []

my\_set = {1, 2, 3}

my\_set.clear()

print(my\_set) # set()

my\_dict = {"a": 1, "b": 2}

my\_dict.clear()

print(my\_dict) # {}

**▍80、合併集合**

使用union()方法，返回一個新集合。

first\_set = {4, 5, 6}

second\_set = {1, 2, 3}

print(first\_set.union(second\_set)) # {1, 2, 3, 4, 5, 6}

還可以使用update()方法，將第二個集合的元素插入到第一個集合中去。

first\_set = {4, 5, 6}

second\_set = {1, 2, 3}

first\_set.update(second\_set)

print(first\_set) # {1, 2, 3, 4, 5, 6}

**▍81、在函數裡輸出結果**

def is\_positive(number):

print("Positive" if number > 0 else "Negative") # Positive

is\_positive(-3)

**▍82、if語句中的多個條件**

math\_points = 51

biology\_points = 78

physics\_points = 56

history\_points = 72

my\_conditions = [math\_points > 50, biology\_points > 50,

physics\_points > 50, history\_points > 50]

if all(my\_conditions):

print("Congratulations! You have passed all of the exams.")

else:

print("I am sorry, but it seems that you have to repeat at least one exam.")

# Congratulations! You have passed all of the exams.

**▍83、在一個if語句中，至少滿足多個條件中的一個**

math\_points = 40

biology\_points = 78

physics\_points = 56

history\_points = 72

my\_conditions = [math\_points > 50, biology\_points > 50,

physics\_points > 50, history\_points > 50]

if any(my\_conditions):

print("Congratulations! You have passed all of the exams.")

else:

print("I am sorry, but it seems that you have to repeat at least one exam.")

# Congratulations! You have passed all of the exams.

**▍84、任何非空字串都為True**

print(bool("Non empty")) # True

print(bool("")) # False

**▍85、任何非空清單、元組、字典都為True**

print(bool([])) # False

print(bool(set([]))) # False

print(bool({})) # False

print(bool({"a": 1})) # True

**▍86、None、False、0都為False**

print(bool(False)) # False

print(bool(None)) # False

print(bool(0)) # False

**▍87、在函數中使用全域變數**

在函數無法直接修改全域變數的值。

string = "string"

def do\_nothing():

string = "inside a method"

do\_nothing()

print(string) # string

可通過修飾符global，修改全域變數的值。

string = "string"

def do\_nothing():

global string

string = "inside a method"

do\_nothing()

print(string) # inside a method

**▍88、計算字串或清單中元素的數量**

使用collections中的Counter計算字串或清單中元素的數量。

from collections import Counter

result = Counter("Banana")

print(result) # Counter({'a': 3, 'n': 2, 'B': 1})

result = Counter([1, 2, 1, 3, 1, 4, 1, 5, 1, 6])

print(result) # Counter({1: 5, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1})

**▍89、檢查2個字串是否為相同**

可以使用Counter()方法。

from collections import Counter

def check\_if\_anagram(first\_string, second\_string):

first\_string = first\_string.lower()

second\_string = second\_string.lower()

return Counter(first\_string) == Counter(second\_string)

print(check\_if\_anagram('testinG', 'Testing')) # True

print(check\_if\_anagram('Here', 'Rehe')) # True

print(check\_if\_anagram('Know', 'Now')) # False

可以使用sorted()方法。

def check\_if\_anagram(first\_word, second\_word):

first\_word = first\_word.lower()

second\_word = second\_word.lower()

return sorted(first\_word) == sorted(second\_word)

print(check\_if\_anagram("testinG", "Testing")) # True

print(check\_if\_anagram("Here", "Rehe")) # True

print(check\_if\_anagram("Know", "Now")) # False

**▍90、使用itertools中的count計算元素的數量**

from itertools import count

my\_vowels = ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']

current\_counter = count()

string = "This is just a sentence."

for i in string:

if i in my\_vowels:

print(f"Current vowel: {i}")

print(f"Number of vowels found so far: {next(current\_counter)}")

輸出如下。

Current vowel: i

Number of vowels found so far: 0

Current vowel: i

Number of vowels found so far: 1

Current vowel: u

Number of vowels found so far: 2

Current vowel: a

Number of vowels found so far: 3

Current vowel: e

Number of vowels found so far: 4

Current vowel: e

Number of vowels found so far: 5

Current vowel: e

Number of vowels found so far: 6

**▍91、對字串或清單的元素進行次數排序**

collections模組的Counter()，預設情況下是不會根據元素的頻率對它們進行排序的。

from collections import Counter

result = Counter([1, 2, 3, 2, 2, 2, 2])

print(result) # Counter({2: 5, 1: 1, 3: 1})

print(result.most\_common()) # [(2, 5), (1, 1), (3, 1)]

map()函數將給定函數應用於可反覆運算物件(清單、元組等)，然後返回結果(map對象)。

**▍92、查找清單中出現頻率最高的元素**

my\_list = ['1', 1, 0, 'a', 'b', 2, 'a', 'c', 'a']

print(max(set(my\_list), key=my\_list.count)) # a

**▍93、copy()和deepcopy()的區別**

淺拷貝: 拷貝父物件，但是不會拷貝物件的內部的子物件。

深拷貝: 拷貝父物件. 以及其內部的子物件。

下面是一個copy()的例子。

first\_list = [[1, 2, 3], ['a', 'b', 'c']]

second\_list = first\_list.copy()

first\_list[0][2] = 831

print(first\_list) # [[1, 2, 831], ['a', 'b', 'c']]

print(second\_list) # [[1, 2, 831], ['a', 'b', 'c']]

這裡是一個deepcopy()的例子。

import copy

first\_list = [[1, 2, 3], ['a', 'b', 'c']]

second\_list = copy.deepcopy(first\_list)

first\_list[0][2] = 831

print(first\_list) # [[1, 2, 831], ['a', 'b', 'c']]

print(second\_list) # [[1, 2, 3], ['a', 'b', 'c']]

**▍94、訪問字典中不存在的鍵時，避免報錯**

如果你想訪問字典一個不存在的鍵，代碼會報錯。

my\_dictonary = {"name": "Name", "surname": "Surname"}

print(my\_dictonary["age"])

錯誤如下。

KeyError: 'age'

可以通過使用defaultdict()，代碼將不會報錯。

from collections import defaultdict

my\_dictonary = defaultdict(str)

my\_dictonary['name'] = "Name"

my\_dictonary['surname'] = "Surname"

print(my\_dictonary["age"])

**▍95、構建反覆運算器**

class OddNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

x = self.a

self.a += 2

return x

odd\_numbers\_object = OddNumbers()

iterator = iter(odd\_numbers\_object)

print(next(iterator)) # 1

print(next(iterator)) # 3

print(next(iterator)) # 5

**▍96、刪除列表的重複項**

my\_set = set([1, 2, 1, 2, 3, 4, 5])

print(list(my\_set)) # [1, 2, 3, 4, 5]

**▍97、列印模組的安裝位置**

import pandas

print(pandas) # <module 'torch' from '/Users/...'

**▍98、使用not in檢查一個值是否在列表中**

odd\_numbers = [1, 3, 5, 7, 9]

even\_numbers = []

for i in range(9):

if i not in odd\_numbers:

even\_numbers.append(i)

print(even\_numbers) # [0, 2, 4, 6, 8]

**▍99、sort()和sorted()的區別**

sort()：對原始列表進行排序

sorted()：返回一個新的排序列表

groceries = ['milk', 'bread', 'tea']

new\_groceries = sorted(groceries)

# new\_groceries = ['bread', 'milk', 'tea']

print(new\_groceries)

# groceries = ['milk', 'bread', 'tea']

print(groceries)

groceries.sort()

# groceries = ['bread', 'milk', 'tea']

print(groceries)

**▍100、使用uuid模組生成唯一ID**

UUID代表唯一識別碼。

import uuid

# 根據主機ID、序號和當前時間生成UUID

print(uuid.uuid1()) # 308490b6-afe4-11eb-95f7-0c4de9a0c5af

# 生成一個隨機UUID

print(uuid.uuid4()) # 93bc700b-253e-4081-a358-24b60591076a

## 1. 易混淆操作

本節對一些 Python 易混淆的操作進行對比。

**1.1 有放回隨機採樣和無放回隨機採樣**

**import** random

random**.**choices(seq, k**=**1) *# 長度為k的list，有放回採樣*

random**.**sample(seq, k) *# 長度為k的list，無放回採樣*

**1.2 lambda 函數的參數**

func **=** **lambda** y: x **+** y *# x的值在函數運行時被綁定*

func **=** **lambda** y, x**=**x: x **+** y *# x的值在函式定義時被綁定*

**1.3 copy 和 deepcopy**

**import** copy

y **=** copy**.**copy(x) *# 只複製最頂層*

y **=** copy**.**deepcopy(x) *# 複製所有嵌套部分*

複製和變數別名結合在一起時，容易混淆：

a **=** [1, 2, [3, 4]]

*# Alias.*

b\_alias **=** a

**assert** b\_alias **==** a **and** b\_alias **is** a

*# Shallow copy.*

b\_shallow\_copy **=** a[:]

**assert** b\_shallow\_copy **==** a **and** b\_shallow\_copy **is** **not** a **and** b\_shallow\_copy[2] **is** a[2]

*# Deep copy.*

**import** copy

b\_deep\_copy **=** copy**.**deepcopy(a)

**assert** b\_deep\_copy **==** a **and** b\_deep\_copy **is** **not** a **and** b\_deep\_copy[2] **is** **not** a[2]

對別名的修改會影響原變數，（淺）複製中的元素是原清單中元素的別名，而深層複製是遞迴的進行複製，對深層複製的修改不影響原變數。

**1.4 == 和 is**

x **==** y *# 兩引用物件是否有相同值*

x **is** y *# 兩引用是否指向同一物件*

**1.5 判斷類型**

type(a) **==** int *# 忽略物件導向設計中的多態特徵*

isinstance(a, int) *# 考慮了物件導向設計中的多態特徵*

**1.6 字串搜索**

str**.**find(sub, start**=None**, end**=None**); str**.**rfind(**...**) *# 如果找不到返回-1*

str**.**index(sub, start**=None**, end**=None**); str**.**rindex(**...**) *# 如果找不到拋出ValueError異常*

**1.7 List 後向索引**

這個只是習慣問題，前向索引時下標從0開始，如果反向索引也想從0開始可以使用~。

print(a[**-**1], a[**-**2], a[**-**3])

print(a[**~**0], a[**~**1], a[**~**2])

## 2. C/C++ 使用者使用指南

不少 Python 的用戶是從以前 C/C++ 遷移過來的，這兩種語言在語法、代碼風格等方面有些不同，本節簡要進行介紹。

**2.1 很大的數和很小的數**

C/C++ 的習慣是定義一個很大的數字，Python 中有 inf 和 -inf：

a **=** float('inf')

b **=** float('-inf')

**2.2 布林值**

C/C++ 的習慣是使用 0 和非 0 值表示 True 和 False， Python 建議直接使用 True 和 False 表示布林值。

a **=** **True**

b **=** **False**

**2.3 判斷為空**

C/C++ 對空指針判斷的習慣是 if (a) 和 if (!a)。Python 對於 None 的判斷是：

**if** x **is** **None**:

**pass**

如果使用 if not x，則會將其他的物件（比如長度為 0 的字串、清單、元組、字典等）都會被當做 False。

**2.4 交換值**

C/C++ 的習慣是定義一個臨時變數，用來交換值。利用 Python 的 Tuple 操作，可以一步到位。

a, b **=** b, a

**2.5 比較**

C/C++ 的習慣是用兩個條件。利用 Python 可以一步到位。

**if** 0 **<** a **<** 5:

**pass**

**2.6 類成員的 Set 和 Get**

C/C++ 的習慣是把類成員設為 private，通過一系列的 Set 和 Get 函數存取其中的值。在 Python 中雖然也可以通過 @property、@setter、@deleter 設置對應的 Set 和 Get 函數，我們應避免不必要的抽象，這會比直接訪問慢 4 - 5 倍。

**2.7 函數的輸入輸出參數**

C/C++ 的習慣是把輸入輸出參數都列為函數的參數，通過指標改變輸出參數的值，函數的返回值是執行狀態，函式呼叫方對返回值進行檢查，判斷是否成功執行。在 Python 中，不需要函式呼叫方進行返回值檢查，函數中遇到特殊情況，直接拋出一個異常。

**2.8 讀文件**

相比 C/C++，Python 讀檔要簡單很多，打開後的檔是一個可反覆運算物件，每次返回一行內容。

**with** open(file\_path, 'rt', encoding**=**'utf-8') **as** f:

**for** line **in** f:

print(line) *# 末尾的\n會保留*

**2.9 檔路徑拼接**

C/C++ 的習慣通常直接用 + 將路徑拼接，這很容易出錯，Python 中的 os.path.join 會自動根據作業系統不同補充路徑之間的 / 或 \ 分隔符號號：

**import** os

os**.**path**.**join('usr', 'lib', 'local')

**2.10 解析命令列選項**

雖然 Python 中也可以像 C/C++ 一樣使用 sys.argv 直接解析命令列選擇，但是使用 argparse 下的 ArgumentParser 工具更加方便，功能更加強大。

**2.11 調用外部命令**

雖然 Python 中也可以像 C/C++ 一樣使用 os.system 直接調用外部命令，但是使用 subprocess.check\_output 可以自由選擇是否執行 Shell，也可以獲得外部命令執行結果。

**import** subprocess

*# 如果外部命令返回值非0，則拋出subprocess.CalledProcessError異常*

result **=** subprocess**.**check\_output(['cmd', 'arg1', 'arg2'])**.**decode('utf-8')

*# 同時收集標準輸出和標準錯誤*

result **=** subprocess**.**check\_output(['cmd', 'arg1', 'arg2'], stderr**=**subprocess**.**STDOUT)**.**decode('utf-8')

*# 執行shell命令（管道、重定向等），可以使用shlex.quote()將參數雙引號引起來*

result **=** subprocess**.**check\_output('grep python | wc > out', shell**=True**)**.**decode('utf-8')

**2.12 不重複造輪子**

不要重複造輪子，Python稱為batteries included即是指Python提供了許多常見問題的解決方案。

## 3. 常用工具

**3.1 讀寫 CSV 文件**

**import** csv

*# 無header的讀寫*

**with** open(name, 'rt', encoding**=**'utf-8', newline**=**'') **as** f: *# newline=''讓Python不將換行統一處理*

**for** row **in** csv**.**reader(f):

print(row[0], row[1]) *# CSV讀到的資料都是str類型*

**with** open(name, mode**=**'wt') **as** f:

f\_csv **=** csv**.**writer(f)

f\_csv**.**writerow(['symbol', 'change'])

*# 有header的讀寫*

**with** open(name, mode**=**'rt', newline**=**'') **as** f:

**for** row **in** csv**.**DictReader(f):

print(row['symbol'], row['change'])

**with** open(name, mode**=**'wt') **as** f:

header **=** ['symbol', 'change']

f\_csv **=** csv**.**DictWriter(f, header)

f\_csv**.**writeheader()

f\_csv**.**writerow({'symbol': xx, 'change': xx})

注意，當 CSV 文件過大時會報錯：\_csv.Error: field larger than field limit (131072)，通過修改上限解決

**import** sys

csv**.**field\_size\_limit(sys**.**maxsize)

csv 還可以讀以 \t 分割的資料

f **=** csv**.**reader(f, delimiter**=**'\t')

**3.2 反覆運算器工具**

itertools 中定義了很多反覆運算器工具，例如子序列工具：

**import** itertools

itertools**.**islice(iterable, start**=None**, stop, step**=None**)

*# islice('ABCDEF', 2, None) -> C, D, E, F*

itertools**.**filterfalse(predicate, iterable) *# 過濾掉predicate為False的元素*

*# filterfalse(lambda x: x < 5, [1, 4, 6, 4, 1]) -> 6*

itertools**.**takewhile(predicate, iterable) *# 當predicate為False時停止反覆運算*

*# takewhile(lambda x: x < 5, [1, 4, 6, 4, 1]) -> 1, 4*

itertools**.**dropwhile(predicate, iterable) *# 當predicate為False時開始反覆運算*

*# dropwhile(lambda x: x < 5, [1, 4, 6, 4, 1]) -> 6, 4, 1*

itertools**.**compress(iterable, selectors) *# 根據selectors每個元素是True或False進行選擇*

*# compress('ABCDEF', [1, 0, 1, 0, 1, 1]) -> A, C, E, F*

序列排序：

sorted(iterable, key**=None**, reverse**=False**)

itertools**.**groupby(iterable, key**=None**) *# 按值分組，iterable需要先被排序*

*# groupby(sorted([1, 4, 6, 4, 1])) -> (1, iter1), (4, iter4), (6, iter6)*

itertools**.**permutations(iterable, r**=None**) *# 排列，返回值是Tuple*

*# permutations('ABCD', 2) -> AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB, DC*

itertools**.**combinations(iterable, r**=None**) *# 組合，返回值是Tuple*

itertools**.**combinations\_with\_replacement(**...**)

*# combinations('ABCD', 2) -> AB, AC, AD, BC, BD, CD*

多個序列合併：

itertools**.**chain(**\***iterables) *# 多個序列直接拼接*

*# chain('ABC', 'DEF') -> A, B, C, D, E, F*

**import** heapq

heapq**.**merge(**\***iterables, key**=None**, reverse**=False**) *# 多個序列按順序拼接*

*# merge('ABF', 'CDE') -> A, B, C, D, E, F*

zip(**\***iterables) *# 當最短的序列耗盡時停止，結果只能被消耗一次*

itertools**.**zip\_longest(**\***iterables, fillvalue**=None**) *# 當最長的序列耗盡時停止，結果只能被消耗一次*

**3.3 計數器**

計數器可以統計一個可反覆運算物件中每個元素出現的次數。

**import** collections

*# 創建*

collections**.**Counter(iterable)

*# 頻次*

collections**.**Counter[key] *# key出現頻次*

*# 返回n個出現頻次最高的元素和其對應出現頻次，如果n為None，返回所有元素*

collections**.**Counter**.**most\_common(n**=None**)

*# 插入/更新*

collections**.**Counter**.**update(iterable)

counter1 **+** counter2; counter1 **-** counter2 *# counter加減*

*# 檢查兩個字串的組成元素是否相同*

collections**.**Counter(list1) **==** collections**.**Counter(list2)

**3.4 帶預設值的 Dict**

當訪問不存在的 Key 時，defaultdict 會將其設置為某個預設值。

**import** collections

collections**.**defaultdict(type) *# 當第一次訪問dict[key]時，會無參數調用type，給dict[key]提供一個初始值*

**3.5 有序 Dict**

**import** collections

collections**.**OrderedDict(items**=None**) *# 反覆運算時保留原始插入順序*

## 4. 高性能程式設計和調試

**4.1 輸出錯誤和警告資訊**

向標準錯誤輸出資訊

**import** sys

sys**.**stderr**.**write('')

輸出警告資訊

**import** warnings

warnings**.**warn(message, category**=UserWarning**)

*# category的取值有DeprecationWarning, SyntaxWarning, RuntimeWarning, ResourceWarning, FutureWarning*

控制警告消息的輸出

$ python **-**W all *# 輸出所有警告，等同於設置warnings.simplefilter('always')*

$ python **-**W ignore *# 忽略所有警告，等同於設置warnings.simplefilter('ignore')*

$ python **-**W error *# 將所有警告轉換為異常，等同於設置warnings.simplefilter('error')*

**4.2 代碼中測試**

有時為了調試，我們想在代碼中加一些代碼，通常是一些 print 語句，可以寫為：

*# 在代碼中的debug部分*

**if** \_\_debug\_\_:

**pass**

一旦調試結束，通過在命令列執行 -O 選項，會忽略這部分代碼：

$ python -0 main.py

**4.3 代碼風格檢查**

使用 pylint 可以進行不少的代碼風格和語法檢查，能在運行之前發現一些錯誤

pylint main.py

**4.4 代碼耗時**

耗時測試

$ python -m cProfile main.py

測試某代碼塊耗時

*# 代碼塊耗時定義*

from contextlib import contextmanager

from time import perf\_counter

@contextmanager

def timeblock**(**label**)**:

tic **=** perf\_counter**()**

try:

yield

finally:

toc **=** perf\_counter**()**

print**(**'%s : %s' % **(**label, toc - tic**))**

*# 代碼塊耗時測試*

with timeblock**(**'counting'**)**:

pass

代碼耗時優化的一些原則

* 專注於優化產生性能瓶頸的地方，而不是全部代碼。
* 避免使用全域變數。區域變數的查找比全域變數更快，將全域變數的代碼定義在函數中運行通常會快 15%-30%。
* 避免使用.訪問屬性。使用 from module import name 會更快，將頻繁訪問的類的成員變數 self.member 放入到一個區域變數中。
* 儘量使用內置資料結構。str, list, set, dict 等使用 C 實現，運行起來很快。
* 避免創建沒有必要的中間變數，和 copy.deepcopy()。
* 字串拼接，例如 a + ':' + b + ':' + c 會創造大量無用的中間變數，':',join([a, b, c]) 效率會高不少。另外需要考慮字串拼接是否必要，例如 print(':'.join([a, b, c])) 效率比 print(a, b, c, sep=':') 低。

## 5. Python 其他技巧

**5.1 argmin 和 argmax**

items **=** [2, 1, 3, 4]

argmin **=** min(range(len(items)), key**=**items**.**\_\_getitem\_\_)

argmax同理。

**5.2 轉置二維列表**

A = [['a11', 'a12'], ['a21', 'a22'], ['a31', 'a32']]

A\_transpose = list(zip(\*A)) # list of tuple

A\_transpose = list(list(col) for col in zip(\*A)) # list of list

**5.3 一維列表展開為二維列表**

A **=** [1, 2, 3, 4, 5, 6]

*# Preferred.*

list(zip(**\***[iter(A)] **\*** 2))

## 列表碾平式

需求： 將[[1,2],[3,4]] 轉換為[1,2,3,4]，具體實現有以下幾種方法

test\_list = [[1,2],[3,4]]

1. from itertools import chain

list(chain.from\_iterable(test\_list))

結果：[1, 2, 3, 4]

2. from itertools import chain

list(chain(\*test\_list))

結果：[1, 2, 3, 4]

3. sum(test\_list, [])

結果：[1, 2, 3, 4]

4. [x for y in test\_list for x in y]

結果：[1, 2, 3, 4]

5. 萬能方法（遞迴）

func = lambda x: [y for t in x for y in func(t)] if type(x) is list else [x]

func(test\_list)

結果：[1, 2, 3, 4]

PS： 專案中，難免會有類似的需求，對於結構嵌套一致的情況，上述的1，2，3，4方法都可以很好的解決(不建議用for迴圈嵌套的方式，那是最low的方法，沒有之一)；對於結構嵌套不一致的情況，第5種方法就派上了用場，其採用了遞迴的思想，堪稱萬能的方法，屢試不爽。專案中，大家可以根據實際應用場景來挑選最適合自己的方法。肚中有糧，心中不慌；大家可以把上述方法都記下來，以做到有的放矢。

## 三目操作符

對於Python的三元運算式，想必大家對if else都不會感到陌生，但是對and or操作想必是另一種感覺了，不過大家對其他語言的? :應該不會陌生，沒錯，這次的主角and or就和? :有著異曲同工之妙。

代碼參考：

1 == 1 and 2 or 3 返回2

1 == 2 and 2 or 3 返回3

PS：習慣於if else的同學偶爾用下and or是不是會給人耳目一新的感覺。

## and or 分開來用

* [x] and 用法如下：

1 and 2 and 3 返回3

1 and 2 and '' 返回''

'' and 2 and 0 返回''

PS：如果都為真則返回最後一個值，如果其中某些值為假，則返回第一個為假的值

* [x] or 用法如下

1 or '' or 0 返回1

'' or 0 or [] 返回[]

PS：如果都為假返回最後一個值，如果其中某些值為真，則返回第一個為真的值

使用場景：在項目中我們經常會有這樣的需求，在將一個字典更新之後還想要返回更新後的字典，這是我們就可以這樣寫：dic = dic1.update(dic2) or dic1

## 列表推導式

需求：將[1,2,3]中的每一項都加1

good: [x+1 for x in [1,2,3]]

bad: def add\_list(goal\_list):

tmp\_list = []

for x in goal\_list:

tmp\_list.append(x+1)

PS: 清單推導式底層是用C實現的，其執行速度要比for迴圈快好多

## vars() 用法

代碼參考：

```

def func(a, b, c):

print vars()

執行func(1, 2, 3)

輸出：{"a":1,"b":2,"c":3}

PS: vars()的值為字典，其鍵值對來源於當前作用域的所有變數。

```

使用場景：在調用他人介面或方法時，需要將傳入的參數列印以記錄日誌，此刻vars()便派上用場了。

## 偏函數之partial

代碼示例：

import functools

def add(a, b):

return a + b

add(4, 2)

6

plus3 = functools.partial(add, 3)

plus5 = functools.partial(add, 5)

plus3(4)

7

plus3(7)

10

plus5(10)

15

實際使用心得：

之前做物件存儲的專案中，我需要同時去調用三個一樣的介面(大部分參數一樣)下面是我的部分代碼，供大家參考

my\_thread = functools.partial(myThread, dic, zone\_id, start, end, customer\_id)

my\_thread1 = my\_thread("day", "gets", "2")

my\_thread1 = my\_thread("day", "original\_traffic", "3")

使用場景：  
當我們需要同時去調用一個函數，並且發現大部分參數一致的時候，便可以採取上述方法，一來代碼簡潔，二來可讀性高。

## Python搭建簡易服務

* Python搭建簡易郵件伺服器：python -m smtpd -n -c DebuggingServer localhost:1025
* Python搭建簡易web伺服器：
  + Python2： python -m SimpleHTTPServer port
  + Python3： python -m http.server port
* Python搭建簡易ftp服務

pip install pyftpdlib

python -m pyftpdlib -p 21

ftp://localhost:21

## 遍歷字典

在專案中遍歷字典是很常見的需求，下面介紹下常用的方法並做下比較：

dic = {'name': 'peter', 'age': 27}

1. for key, value in dic.items():

print key, value

2. for key, value in dic.iteritems():

print key, value

PS:  
iteritems和items的區別在於iteritems採用了生成器的原理，只有在需要的時候才會把值生成，其之間的區別類似於range和 xrange；readline和 xreadline

## 記憶體管理

Python的記憶體管理主要分為引用計數和垃圾回收機制兩大部分，且看下麵代碼：

* [ ] 記憶體分配：

a = 1

b = 1

a is b True

---------------------

a = 1000

b = 1000

a is b False

PS: 在Python中，整數和短小的字元，Python都會緩存這些物件，以便重複使用。當我們創建多個等於1的引用時，實際上是讓這些引用指向了同一個物件。

* [ ] 引用計數：

在Python中，所謂引用計數(reference count)是指所有指向該物件的引用的總數;

我們可以使用sys包中的getrefcount(),來查看某個物件的引用計數。需要注意的是，當使用該函數查看某個物件的引用計數時，實際上是臨時創建了該物件的一個新的引用，所有使用getrefcount()所得到的結果，會比期望的值多1。

from sys import getrefcount

aa = 'test refcount' print(getrefcount(a))

bb = aa

print(getrefcount(a))

PS: 由於上述原因，兩個getrefcount()將返回2和3，並不是期望的1和2.

* [ ] 引用減少

引用減少大致分為兩類：

* + 指向該物件的引用指向了其他物件

from sys import getrefcount

aa = 'test refcount'

bb = aa

print(getrefcount(aa)) 3

bb = 1

print(getrefcount(aa)) 2

* + 使用del關鍵字顯示的刪除某個引用

from sys import getrefcount

aa = 'test refcount'

bb = aa

print(getrefcount(aa)) 3

del bb

print(getrefcount(aa)) 2

* [ ] 垃圾回收

不斷的創建物件，如果不及時銷毀的話，那Python的體積會越來越大，再大的記憶體也會有耗完的時候；不用像C語言那樣，需要手動的去管理記憶體、Python已經幫我們做好了(Python的垃圾回收機制)，你只需要去關心你的業務邏輯即可，其他的都交給Python來處理。

從原理上講，當Python中某個對象的引用計數降為0時，該物件就應該被回收。但是頻繁的啟動垃圾回收機制畢竟是個很耗時的問題；因此Python只有在特定條件下(當Python中被分配物件和取消分配物件的次數之間的差值達到某個閾值時)，Python會自動啟動垃圾回收機制。

我們可以通過gc模組的get\_threshold()方法，查看該閾值：

import gc

print(gc.get\_threshold())

該方法會返回(700, 10, 10)，後面的倆10是與分代回收相關的，稍後講解。700便是垃圾回收機制啟動的閾值。可以通過gc模組中的set\_threshold()方法重新設定該值。

當然了，我們也可以手動啟動垃圾回收機制，使用gc.collect()即可。

* [ ] 分代回收

Python同時採用了分代回收的機制，設想一下：存活越久的物件、越不可能是垃圾物件。程式在運行時，往往會產生大量的臨時物件，程式結束之後，這些臨時物件的生命週期也就隨之告一段落。但有一些對象會被長期佔用，垃圾回收機制在啟動的時候會減少掃描到他們的頻率。

Python將所有物件分為0，1，2三代。所有新創建的物件都是0代，當垃圾回收機制在啟動多次0代機制並掃描到他們的時候，這些物件如果依然存活在記憶體的話，他們就會被歸入下一代對象，以此類推。

剛才上面所提到的(700, 10, 10)三個參數後面的倆10所代表的意思是：每10次0代垃圾回收，會配合1次1代的垃圾回收；而每10次1代的垃圾回收會配合1次的2代垃圾回收。

當然我們同樣可以使用set\_threshold()來調整此策略的比例，比如對1代物件進行更頻繁的掃描。

import gc

gc.set\_threshold(700, 5, 10)

## 新式類、經典類

* [ ] 新式類: 顯示的繼承了object的類

class A(object):

attr = 1

class B(A):

pass

class C(A):

attr = 3

class D(B, C):

pass

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

d = D()

print 'attr = ', d.attr # attr = 3

* [ ] 經典類：沒有繼承自object的類

class A():

attr = 1

class B(A):

pass

class C(A):

attr = 3

class D(B, C):

pass

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

d = D()

print 'attr = ', d.attr # attr = 1

PS: 通過以上代碼的輸出結果可以看出，新式類會廣度搜索，也就是一層層的向上搜索；經典類是深度優先，即遇到一個超類點就向上搜索。

## 裝飾器

Python的裝飾器被稱為Python的語法糖，哪裡需要粘哪裡。

代碼示例：

@makeh1

@makeeitalic

def say():

return 'Peter'

我們希望輸出結果為：<h1><i>Peter</i></h1>

去看看[官方文檔](https://link.zhihu.com/?target=https%3A//link.juejin.im/%3Ftarget%3Dhttps%253A%252F%252Fdocs.python.org%252F2%252Freference%252Fcompound_stmts.html%2523function" \t "_blank)，答案就看下面：

def makeh1(func):

def wrp():

return "<h1>" + func() + "</h1>"

return wrp

def makeeitalic(func):

def wrp():

return "<i>" + func() + "</i>"

return wrp

@makeh1

@makeeitalic

def say():

return 'Hello Peter'

print say()

輸出：<h1><i>Hello Peter</i></h1>

實際應用場景：

使用過django的小夥伴想必都用過login\_required裝飾器，但是如果用戶沒登錄的話login\_required會重定向到登錄頁面；在做web開發的過程中，我們會經常用  
ajax非同步提交資料到後臺，這時如果再繼續使用原有的login\_required裝飾器肯定是不行了(該裝飾器不會重定向到登錄頁面，ajax也沒有任何返回結果)，下麵我們改變下原有代碼：

from django.shortcuts import HttpResponse

import json

def is\_authenticat(func):

def wrp(req, \*\*kwargs):

if req.user.is\_authenticated():

return func(req, \*\*kwargs)

else:

json\_str = {'status': 0, 'msg': u'請登錄'}

return HttpResponse(json.dumps(json\_str), content\_type='application/json')

return wrp

上述代碼便很好的解決了問題，也算是對Python裝飾器的一個很好的使用場景。

## all or any

Python 語言如此流行的眾多原因之一，是因為它具有很好的可讀性和表現力。

人們經常開玩笑說 Python 是可執行的偽代碼。當你可以像這樣寫代碼時，就很難反駁。

x = [True, True, False]

if any(x):

print("至少有一個True")

if all(x):

print("全是True")

if any(x) and not all(x):

print("至少一個True和一個False")

複製代碼

## bashplotlib

你有沒有想過在控制台中繪製圖形嗎？

Bashplotlib 是一個 Python 庫，他能夠幫助我們在命令列(粗曠的環境)中繪製資料。

# 模組安裝

pip install bashplotlib

# 繪製實例

import numpy as np

from bashplotlib.histpgram import plot\_hist

arr = np.ramdom.normal(size=1000, loc=0, scale=1)

plot\_hist(arr, bincount=50)

複製代碼

## collections

Python 有一些很棒的默認資料類型，但有時它們的行為並不完全符合你的期望。

幸運的是，Python 標準庫提供了 collections 模組[1]。這個方便的附加元件為你提供了更多的資料類型。

from collections import OrderedDict, Counter

# 記住鍵的添加順序！

x = OrderedDict(a=1, b=2, c=3)

# 統計每個字元出現的頻率

y = Counter("Hello World!")

複製代碼

## dir

有沒有想過如何查看 Python 物件內部，查看它具有哪些屬性？在命令列中輸入：

dir()

dir("Hello World")

dir(dir)

複製代碼

## emoji

emoji[3] 是日本在無線通訊中所使用的視覺情感符號，繪指圖畫，文字指的則是字元，可用來代表多種表情，如笑臉表示笑、蛋糕表示食物等。在中國大陸，emoji通常叫做“小黃臉”，或者直稱emoji。

# 安裝模組

pip install emoji

# 做個嘗試

from emoji import emojize

print(emojize(":thumbs\_up:"))

複製代碼

## fromfuture import

Python 流行的結果之一，總是有新版本正在開發中。新版本意味著新功能 —— 除非你的版本已過時。

不過不要擔心。使用該\_\_future\_\_模組[4]可以説明你用Python的未來版本導入功能。從字面上看，這就像是時間旅行、魔法或其他東西

from \_\_future\_\_ import print\_function

print("Hello World!")

複製代碼

## geogy

地理，對大多數程式師來說是一個具有挑戰性的領域。在獲取地理資訊或者繪製地圖時，也會遇到不少問題。這個geopy 模組[5]讓地理相關內容變得非常容易。

pip install geopy

複製代碼

它通過抽象一系列不同地理編碼服務的 API 來工作。通過它，你能夠獲得一個地方的完整街道位址、緯度、經度甚至海拔高度。

還有一個有用的距離類。它以最好的測量單位計算兩個位置之間的距離。

from geopy import GoogleV3

place = "221b Baker Street, London"

location = GoogleV3().geocode(place)

print(location.address)

print(location.location)

複製代碼

## howdoi

當你使用terminal終端程式設計時，通過在遇到問題後會在StackOverflow上搜索答案，完後會回到終端繼續程式設計，此時有時會不記得你之前查到的解決方案，此時需要重新查看StackOverflow，但又不想離開終端，那麼此時你需要用到這個有用的命令列工具howdoi[6]。

pip install howdoi

複製代碼

無論你有什麼問題，都可以問它，它會盡力回復。

howdoi vertical align css

howdoi for loop in java

howdoi undo commits in git

複製代碼

但請注意——它會在 StackOverflow 的最佳答案中抓取代碼。它可能並不總是提供最有用的資訊......

howdoi exit vim

**1、如何使用 print 輸出日誌**

初學者喜歡使用 print 來調試代碼，並記錄程式運行過程。

但是 print 只會將內容輸出到終端上，不能持久化到日誌檔中，並不利於問題的排查。

如果你熱衷於使用 print 來調試代碼（雖然這並不是最佳做法），記錄程式運行過程，那麼下面介紹的這個 print 用法，可能會對你有用。

Python 3 中的 print 作為一個函數，由於可以接收更多的參數，所以功能變為更加強大，指定一些參數可以將 print 的內容輸出到日誌檔中。

>>> with open('test.log', mode='w') as f:

... print('hello, python', file=f, flush=True)

>>> exit()

$ cat test.log

hello, python

**2、規則運算式基本配方**

import re

pattern = re.compile(r”\d\d”)

print(re.search(pattern,"Let's find the number 23").group())

# or

print(re.findall(pattern, “Let's find the number 23”))[0]

# Outputs

'23'

'23'

Regex 對於許多 python 管道來說都是必須的，所以記住***核心Regex方法***很有用處。

**3、將嵌套 for 迴圈寫成單行**

我們經常會如下這種嵌套的 for 迴圈代碼

list1 = range(1,3)

list2 = range(4,6)

list3 = range(7,9)

for item1 in list1:

for item2 in list2:

for item3 in list3:

print(item1+item2+item3)

這裡僅僅是三個 for 迴圈，在實際編碼中，有可能會有更層。這樣的代碼，可讀性非常的差，很多人不想這麼寫，可又沒有更好的寫法。

這裡介紹一種常用的寫法，***使用 itertools 這個庫***來實現更優雅易讀的代碼。

from itertools import product

list1 = range(1,3)

list2 = range(4,6)

list3 = range(7,9)

for item1,item2,item3 in product(list1, list2, list3):

print(item1+item2+item3)

輸出：

$ python demo.py

12

13

13

14

13

14

14

15

**4、如何在運行狀態查看原始程式碼？**

查看函數的原始程式碼，我們通常會使用 IDE 來完成。比如在 PyCharm 中，你可以 Ctrl + 滑鼠點擊 進入函數的原始程式碼。

那如果沒有 IDE 呢？想使用一個函數時，如何知道這個函數需要接收哪些參數呢？

這時可以使用***inspect 來代替 IDE***幫助你完成這些事

# demo.py

import inspect

def add(x, y):

return x + y

print("===================")

print(inspect.getsource(add))

運行結果：

$ python demo.py

===================

def add(x, y):

return x + y

這裡我想先說一個點：不管你是通過各種管道學Python還是自學？還是說你大學在學習，甚至於說有別的程式設計的基礎，一定要注重一個東西：***完整的知識體系***。對於每一個自學的人，按照這個體系去打好基礎，你未來的路會走得更穩重。

適用人群：零基礎 / 基礎不紮實者，學Python都從這裡開始

[零基礎學Python有什麼建議？687 贊同 · 7 評論回答](https://www.zhihu.com/question/451604793/answer/1850553245" \t "_blank)

**5、最快查看包搜索路徑的方式**

當你使用 import 導入一個包或模組時，Python 會去一些目錄下查找，而這些目錄是有優先順序順序的，正常人會使用 sys.path 查看。

>>> import sys

>>> from pprint import pprint

>>> pprint(sys.path)

['',

'/usr/local/Python3.7/lib/python37.zip',

'/usr/local/Python3.7/lib/python3.7',

'/usr/local/Python3.7/lib/python3.7/lib-dynload',

'/home/wangbm/.local/lib/python3.7/site-packages',

'/usr/local/Python3.7/lib/python3.7/site-packages']

>>>

有沒有更快的方式呢？一行命令即可解決！

[wangbm@localhost ~]$ python3 -m site

sys.path = [

'/home/wangbm',

'/usr/local/Python3.7/lib/python37.zip',

'/usr/local/Python3.7/lib/python3.7',

'/usr/local/Python3.7/lib/python3.7/lib-dynload',

'/home/wangbm/.local/lib/python3.7/site-packages',

'/usr/local/Python3.7/lib/python3.7/site-packages',

]

USER\_BASE: '/home/wangbm/.local' (exists)

USER\_SITE: '/home/wangbm/.local/lib/python3.7/site-packages' (exists)

ENABLE\_USER\_SITE: True

從輸出你可以發現，這個列的路徑會比 sys.path 更全，它包含了使用者環境的目錄。

**6、如何快速計算函數執行時間**

計算一個函數的執行時間，有人可能會這樣寫：

import time

start = time.time()

# run the function

end = time.time()

print(end-start)

你看看你為了計算函數執行時間，寫了幾行代碼了？

有沒有可以更方便計算的呢？

***內置模組叫 timeit！***使用它，只用一行代碼即可

import time

import timeit

def run\_sleep(second):

print(second)

time.sleep(second)

# 只用這一行

print(timeit.timeit(lambda :run\_sleep(2), number=5))

運行結果

2

2

2

2

2

10.020059824

**7、partial 函數**

from functools import partial

def multiply(x,y):

return x\*y

dbl = partial(multiply,2)

print(dbl)

print(dbl(4))

# Outputs

functools.partial(<function multiply at 0x7f16be9941f0>, 2)

8

在這裡，我們創建一個函數，它複製另一個函數，但使用的參數比原始函數少，這樣就可以使用它將該參數應用於多個不同的參數。

**8、使用 hasattr() 內置方法獲取object屬性**

class SomeClass:

def \_\_init\_\_(self):

self.attr1 = 10

def attrfunction(self):

print("Attreibute")

hasattr(SomeClass, "attrfunction")

# Output

True

**9、使用 defaultdict 和 lambda 函數創建字典**

from collections import defaultdict

import numpy as np

q = defaultdict(lambda: np.zeros(5))

# Example output

In : q[0]

Out: array([0., 0., 0., 0., 0.])

defaultdicts不會引發KeyError，任何不存在的鍵都會獲取預設工廠返回的值。  
在上述代碼，預設工廠是一個lambda函數，它為給定的任何鍵返回一個預設NumPy陣列，其中包含5個零。

**10、使用集合從兩個清單中獲得差異**

list1 = [1,2,3,4,5]

list2 = [3,4,5]

print(list(set(list1) — set(list2)))

# or

print(set(lista1).difference(set(lista2)))

# Outputs

[1,2]

{1,2}

在這裡，集合有助於獲得兩個python列表之間的差異，這兩個列表既是一個清單，也是一個集合。

**11、使用isinstance（）檢查變數是否為給定類型**

isinstance(1, int)

#Output

True

**12、使用map（）列印清單中的數位**

一種比迴圈列印清單內容更快更有效的方法

list1 = [1,2,3]

list(map(print, list1))

# Output

1

2

3

**13、使用.join（）方法格式化datetime日期**

from datetime import datetime

date = datetime.now()

print("-".join([str(date.year), str(date.month), str(date.day)])

**14、將兩個具有相同規則的清單隨機化**

import numpy as np

x = np.arange(100)

y = np.arange(100,200,1)

idx = np.random.choice(np.arange(len(x)), 5, replace=False)

x\_sample = x[idx]

y\_sample = y[idx]

print(x\_sample)

print(y\_sample)

# Outputs

array([68, 87, 41, 16, 0])

array([168, 187, 141, 116, 100])

**15、對輸入的字串“消毒”**

對使用者輸入的內容“消毒”，這問題幾乎適用於你編寫的所有程式。

通常將字元轉換為小寫或大寫就足夠了，有時還可以使用規則運算式來完成工作，但是對於複雜的情況，還有更好的方法：

user\_input = "This\nstring has\tsome whitespaces...\r\n"

character\_map = {

ord('\n') : ' ',

ord('\t') : ' ',

ord('\r') : None

}

user\_input.translate(character\_map) # This string has some whitespaces... "

在上述代碼，可以看到空格字元“ \n”和“ \t”被單個空格替換了，而“ \r”則被完全刪除。

這是一個簡單的示例，但是我們可以更進一步，使用unicodedata 庫及其 combining() 函數，來生成更大的重映射表（remapping table），並用它來刪除字串中所有的重音。

**16、反轉字串**

編寫一些代碼來反轉字串

def reverse\_string(string):

result=""

for c in range(len(string),-1,-1):

result = result + string[c]

return result

看起來有點亂吧， 用另一種方式來表達：

def reverse\_string(string):

result = [ string[c] for c in range(len(string),-1,-1)]

return "".join(result)

看起來不錯吧，其實使用切片的方法可以更好，方法如下：

def reverse\_string(string):

return string[::-1]

**17、將數位與數位進行求和**

將數位與數位進行求和，這是一個非常簡單的問題，我們可以用傳統方法解決這個問題：

def sum\_a\_num(num):

sum = 0

while num > 0:

sum+= num%10

num//=10

return sum

這是可以的，但在緊張的情況下，最終可能會與運運算元發生拼寫錯誤或錯誤，最終得到錯誤的結果，並花費數小時嘗試調試代碼。

想要避免這種情況，有一個更好的方法做到這一點：

def sum\_a\_num(num):

return sum(list(map(int,str(num))))

上述代碼所做的是：

* 將 num 轉換為字串；
* map() 函數在字串上遍數，並將每個字元轉換為整數；
* list() 函數將映射物件轉換為清單，然後求和；

num = 2367

str(num) = '2367'

list(map(int,str(num))) = [2,3,6,7]

sum(list(map(int(str(num)))) = 18

**18、僅支援關鍵字參數（kwargs）的函數**

當需要函數提供（強制）更清晰的參數時，創建僅支援關鍵字參數的函數，可能會挺有用：

def test(\*, a, b):

pass

test("value for a", "value for b") # TypeError: test() takes 0 positional arguments...

test(a="value", b="value 2") # Works...

如上所見，可以在關鍵字參數之前，放置單個 \* 參數來輕鬆解決此問題，如果我們將位置參數放在 \* 參數之前，則顯然也可以有位置參數。

**19、使用slice函數命名切片**

使用大量硬編碼的索引值會很快搞亂維護性和可讀性，一種做法是對所有索引值使用常量，但是我們可以做得更好：

# ID First Name Last Name

line\_record = "2 John Smith"

ID = slice(0, 8)

FIRST\_NAME = slice(9, 21)

LAST\_NAME = slice(22, 27)

name = f"{line\_record[FIRST\_NAME].strip()} {line\_record[LAST\_NAME].strip()}"

# name == "John Smith"

在此例中，我們可以避免神秘的索引，方法是先使用 slice 函數命名它們，然後再使用它們，還可以通過 .start、.stop和 .stop 屬性，來瞭解 slice 物件的更多資訊。

**20、在運行時提示使用者輸入密碼**

許多命令列工具或腳本需要用戶名和密碼才能操作。因此，如果你碰巧寫了這樣的程式，你可能會發現 getpass 模組很有用：

import getpass

user = getpass.getuser()

password = getpass.getpass()

# Do Stuff...

這個非常簡單的包通過提取當前用戶的登錄名，可以提示使用者輸入密碼，但是須注意，並非每個系統都支援隱藏密碼。Python 會嘗試警告你，因此切記在命令列中閱讀警告資訊。

**21、用\_\_slots\_\_節省記憶體**

如果你曾經編寫過一個程式，該程式創建了某個類的大量實例，那麼你的程式突然就會需要大量記憶體。那是因為 Python 使用字典來表示類實例的屬性，這能使其速度變快，但記憶體不是很高效。

通常這不是個問題，但是，如果你的程式遇到了問題，你可以嘗試使用\_\_slots\_\_ ：

class Person:

\_\_slots\_\_ = ["first\_name", "last\_name", "phone"]

def \_\_init\_\_(self, first\_name, last\_name, phone):

self.first\_name = first\_name

self.last\_name = last\_name

self.phone = phone

這裡發生的是，當我們定義\_\_slots\_\_屬性時，Python 使用固定大小的小型陣列，而不是字典，這大大減少了每個實例所需的記憶體。

使用\_\_slots\_\_還有一些缺點——我們無法聲明任何新的屬性，並且只能使用在\_\_slots\_\_中的屬性。  
同樣，帶有\_\_slots\_\_的類不能使用多重繼承。

**22、限制CPU和記憶體使用量**

如果不是想優化程式記憶體或 CPU 使用率，而是想直接將其限制為某個固定數字，那麼 Python 也有一個庫能做到：

import signal

import resource

import os

# To Limit CPU time

def time\_exceeded(signo, frame):

print("CPU exceeded...")

raise SystemExit(1)

def set\_max\_runtime(seconds):

# Install the signal handler and set a resource limit

soft, hard = resource.getrlimit(resource.RLIMIT\_CPU)

resource.setrlimit(resource.RLIMIT\_CPU, (seconds, hard))

signal.signal(signal.SIGXCPU, time\_exceeded)

# To limit memory usage

def set\_max\_memory(size):

soft, hard = resource.getrlimit(resource.RLIMIT\_AS)

resource.setrlimit(resource.RLIMIT\_AS, (size, hard))

我們可以看到兩個選項，可設置最大 CPU 執行時間和記憶體使用上限。

對於 CPU 限制，我們首先獲取該特定資源（RLIMIT\_CPU）的軟限制和硬限制，然後通過參數指定的秒數和先前獲取的硬限制來設置它。

最後，如果超過 CPU 時間，我們將註冊令系統退出的信號。至於記憶體，我們再次獲取軟限制和硬限制，並使用帶有 size 參數的setrlimit 和獲取的硬限制對其進行設置。

**下面重點介紹幾個和反覆運算相關的使用技巧，可以方便提升大家的工作效率。**

**很多人學Python搞不清楚方向，不同目的，你學習的側重點和難易程度都不同，必須要有針對性、選擇性地學！這樣也能提高你自己的學習效率。**

[Python學習的方法687 贊同 · 7 評論回答](https://www.zhihu.com/question/451604793/answer/1850553245" \t "_blank)

Iterables是一個需要我們牢記的概念，因為接下來我們展示的許多技巧都使用itertools包。itertools模組提供了一些函數，用於接收Iterable物件，而不僅僅是列印逐個對象。

iterables的示例包括：

* 所有序列類型（如list、str和tuple）
* 一些非序列類型，如dict、檔物件以及類的實現中定義了\_\_iter\_\_()方法

在工作學習中，我們經常會需要使用一個簡單的函數來實現從一個list來生成新的list、set或dict，此時我們就會用到iterables概念。

**23、舉例來說：**

生成List：

names = ['John', 'Bard', 'Jessica' 'Andres']

lower\_names = [name.lower() for name in names]

生成Set：

names = ['John', 'Bard', 'Jessica' 'Andres']

lower\_names = {name.lower() for name in names}

生成Dict：

names = ['John', 'Bard', 'Jessica' 'Andres']

lower\_names = {name:name.lower() for name in names}

個人建議：

僅當for語句、函式呼叫和方法調用的數量較少時使用

**24、有時我們需要獲得兩個清單物件之間的所有可能組合，我們可能首先想到的是：**

l1 = [1, 2, 3]

l2 = [4, 5, 6]

combinations = []

for e1 in l1:

for e2 in l2:

combinations.append((e1, e2))

或者簡化一下

combinations = [(e1, e2) for e1 in l1 for e2 in l1]

上述實現已經很簡潔了，但標準庫itertools提供product函數，從而提供了相同的結果。

from itertools import product

l1 = [1, 2, 3]

l2 = [4, 5, 6]

combinatios = product(l1, l2)

**25、假設有一個元素清單，我們需要在每對相鄰元素之間比較或應用一些操作，這有時稱為2個元素的滑動視窗。**

可以採用以下方式：

from itertools import tee

from typing import Iterable

def window2(iterable: Iterable):

it, offset = tee(iter(iterable))

next(offset)

return zip(it, offset)

l = [1, 2, 3, 4, 5, 6]

dd = window2(l)

for a in dd:

print(a)

運行結果：

(1, 2)

(2, 3)

(3, 4)

(4, 5)

(5, 6)

**26、當需要一個類來存儲資訊，又覺得創建一個類並定義其\_\_init\_\_()函數太麻煩，不妨選擇使用dataclass。**

from dataclasses import dataclass

@dataclass

class Person:

name: str

age: int

address: str

上述代碼創建了一個具有預設構造函數的類，該類以與聲明相同的順序接收相應欄位的賦值。

person = Person(name='John', age=12, address='nanjing street')

dataclass的另一個優點是，預設情況下，會生成特殊方法，如\_\_str\_\_、**repr**、\_\_eq\_\_等。

注意：dataclasses構造物件時並不執行資料類型的檢查

**27、假如我們有一個dataclass，需要驗證輸入資料是否符合類型注釋。**

在這種情況下，安裝協力廠商套裝軟體pydantic並將from dataclasses import dataclass 替換為 from pydantic.dataclasses import dataclass 即可。

from pydantic.dataclasses import dataclass

@dataclass

class Person:

name: str

age: int

address: str

這將生成一個類，該類具有根據成員變數聲明的類型進行輸入資料的解析和類型驗證。Pydantic在運行時強制執行類型提示，並在資料無效時提供友好的錯誤提醒。

**28、如果我們對兩個list中的元素對做相應的函數處理**

我們最容易想到的方法：

l1 = [1, 2, 3]

l2 = [4, 5, 6]

for (e1, e2) in zip(l1, l2):

f(e1, e2)

但使用函數map可以讓代碼更加簡潔一些；

l1 = [1, 2, 3]

l2 = [4, 5, 6]

map(f, l1, l2)

**29、從一個list中隨機選擇一個元素**

此時我們使用random.choice

from random import choice

l = [1, 2, 3]

random = choice(l)

如果需要隨機選擇多個元素呢？當然是使用***random.choices***

from random import choices

l = [1, 2, 3, 4, 5]

random\_elements = choices(l, k=3)

代碼中的參數k為我們隨機選擇元素的個數；

**30、跳過可反覆運算物件的開始**

有時候你必須處理某些檔，它們以可變數量的不需要的行（例如注釋）為開頭。

string\_from\_file = """

// Author: ...

// License: ...

//

// Date: ...

Actual content...

"""

import itertools

for line in itertools.dropwhile(lambda line:line.startswith("//"), string\_from\_file.split("\n")):

print(line)

## 一、根據條件在序列中篩選資料

* 假設有一個數字清單 data, 過濾列表中的負數

data = [1, 2, 3, 4, -5]

# 使用列表推導式

result = [i for i in data if i >= 0]

# 使用 fliter 過濾函數

result = filter(lambda x: x >= 0, data)

* 學生的數學分數以字典形式存儲，篩選其中分數大於 80 分的同學

from random import randint

d = {x: randint(50, 100) for x in range(1, 21)}

r = {k: v for k, v in d.items() if v > 80}

## 二、對字典的鍵值對進行翻轉

* 使用 zip() 函數

zip() 函數用於將可反覆運算的物件作為參數，將物件中對應的元素打包成一個個元組，然後返回由這些元組組成的列表。

from random import randint, sample

s1 = {x: randint(1, 4) for x in sample("abfcdrg", randint(1, 5))}

d = {k: v for k, v in zip(s1.values(), s1.keys())}

## 三、統計序列中元素出現的頻度

* 某隨機序列中，找到出現次數最高的3個元素，它們出現的次數是多少

**方法1:**

# 可以使用字典來統計，以清單中的資料為鍵，以出現的次數為值

from random import randint

# 構造隨機序列

data = [randint(0, 20) for \_ in range(30)]

# 清單中出現數字出現的次數

d = dict.fromkeys(data, 0)

for v in d:

d[v] += 1

**方法2：**

# 直接使用 collections 模組下面的 Counter 物件

from collections import Counter

from random import randint

data = [randint(0, 20) for \_ in range(30)]

c2 = Counter(data)

# 查詢元素出現次數

c2[14]

# 統計頻度出現最高的3個數

c2.most\_common(3)

* 對某英文文章單詞進行統計，找到出現次數最高的單詞以及出現的次數

import re

from collections import Counter

# 統計某個文章中英文單詞的詞頻

with open("test.txt", "r", encoding="utf-8") as f:

d = f.read()

# 所有的單字清單

total = re.split("\W+", d)

result = Counter(total)

print(result.most\_common(10))

## 四、根據字典中值的大小，對字典中的項進行排序

* 比如班級中學生的數學成績以字典的形式存儲，請按數學成績從高到底進行排序

**方法1:**

# 利用 zip 將字典轉化為元組，再用 sorted 進行排序

from random import randint

data = {x: randint(60, 100) for x in "xyzfafs"}

sorted(data)

data = sorted(zip(data.values(), data.keys()))

**方法2:**

# 利用 sorted 函數的 key 參數

from random import randint

data = {x: randint(60, 100) for x in "xyzfafs"}

data.items()

sorted(data.items(), key=lambda x: x[1])

## 五、在多個字典中找到公共鍵

* 實際場景：在足球聯賽中，統計每輪比賽都有進球的球員

第一輪：{"C羅": 1, "蘇亞雷斯":2, "托雷斯": 1..}

第二輪：{"內瑪律": 1, "梅西":2, "姆巴佩": 3..}

第三輪：{"姆巴佩": 2, "C羅":2, "內瑪律": 1..}

from random import randint, sample

from functools import reduce

# 模擬隨機的進球球員和進球數

s1 = {x: randint(1, 4) for x in sample("abfcdrg", randint(1, 5))}

s2 = {x: randint(1, 4) for x in sample("abfcdrg", randint(1, 5))}

s3 = {x: randint(1, 4) for x in sample("abfcdrg", randint(1, 5))}

# 首先獲取字典的 keys，然後取每輪比賽 key 的交集。由於比賽輪次數是不定的，所以使用 map 來批量操作

# map(dict.keys, [s1, s2, s3])

# 然後一直累積取其交集，使用 reduce 函數

reduce(lambda x, y: x & y, map(dict.keys, [s1, s2, s3]))

## List

Python 的清單 List 基本就是其它語言的 Array.

### Initialization 初始化

List 的初始化一般用 List comprehension，往往能一行解決問題

*# 1d array*

l **=** [0 **for** \_ **in** range(len(array)]

*# or*

l **=** [0] **\*** len(array)

*# 2d*

l **=** [[0] **for** i **in** range(cols) **for** j **in** range(rows)]

### Start from the behind

你可以輕鬆從後往前訪問：

lastElement = l[-1]

lastTwo = l[-2:]

for i in range(0, -10, -1)

# 0, -1, -2, -3, -4, -5, -6, -7, -8, -9

### copy 複製

### shallow copy 淺拷貝

l2 = l1[:]

# or

l2 = l1.copy()

淺複製的問題在於，如果 l1 內部還有 list，那麼這種嵌套的索引不能被複製，比如：

a = [1, 2, [3, 4]]

b = a[:]

a[2].append(5)

print(b)

# [1, 2, [3, 4, 5]]

### deep copy 深拷貝

所以如果要做深拷貝，要節制自帶庫 copy

import copy

copy.deepcopy()

### enumerate 枚舉

當我們需要枚舉一個陣列並同時獲得值與 index 的時候可以使用：

l = ["a", "b", "c"]

for i, v in enumerate(l):

print(i, v)

# 0 a

# 1 b

# 2 c

### zip

zip 本意就是拉鍊，可以想像成將兩個陣列像拉鍊一樣挨個聚合：

>>> x = [1, 2, 3]

>>> y = [4, 5, 6]

>>> zipped = zip(x, y)

>>> list(zipped)

[(1, 4), (2, 5), (3, 6)]

### reduce

reduce 可以分別對相鄰元素使用同一種計算規則，同時每一步結果作為下一步的參數，很典型的函數式程式設計用法。

# importing functools for reduce()

import functools

# initializing list

lis = [ 1, 3, 5, 6, 2, ]

# using reduce to compute sum of list

print ("The sum of the list elements is : ",end="")

print (functools.reduce(lambda a,b : a+b,lis))

# The sum of the list elements is : 17

### map

可以將參數一一映射來計算， 比如

date = "2019-8-15"

Y, M, D = map(int, date.split('-'))

# Y = 2019, M = 8, D = 15

## deque

list 刪除末尾的操作是O(1)的，但是刪除頭操作就是O(n)，這時候我們就需要一個雙端佇列 deque。首尾的常規操作為：

* append，添加到末尾
* appendleft, 添加到開頭
* pop, 剔除末尾
* popleft，移除開頭

## sorted

list 自身有自帶的 sort(), 但是它不返回新的 list. sorted 能返回一個新的 list, 並且支持傳入參數reverse。

比如我們有一個 tuple 的陣列，我們想按照 tuple 的第一個元素進行排序：

l1 = [(1,2), (0,1), (3,10) ]

l2 = sorted(l1, key=lambda x: x[0])

# l2 = [(0, 1), (1, 2), (3, 10)]

這裡的 key 允許傳入一個自訂參數，也可以用自帶函數進行比較，比如在一個 string 陣列裡只想比較小寫，可以傳入key=str.lower

l1 = ["banana","APPLE", "Watermelon"]

l2 = sorted(l1, key=str.lower)

print(l2)

# ['APPLE', 'banana', 'Watermelon']

### lambda

你注意到我們在上面使用了 lambda 來定義一個匿名函數，十分方便。如果你熟悉其它語言類似 JS 的話，可以把它理解成一個 callback 函數，參數名一一對應就行。

### cmp\_to\_key

在 python3 中，sorted 函數取消了自帶的cmp函數，需要借助functools 庫中的 cmp\_to\_key來做比較。  
比如如果要按照陣列元素的絕對值來排序：

from functools import cmp\_to\_key

def absSort(arr):

newarr = sorted(arr, key = cmp\_to\_key(sortfunc))

return newarr

def sortfunc(a, b):

if abs(a) < abs(b):

return -1

elif abs(a) > abs(b):

return 1

else:

return a - b

## set

set 的查找操作複雜度為O(1)，有時候可以替代dict 來存儲中間過程。

* add : set 的添加是 add 不是append
* remove vs discard: 都是刪除操作，區別在於remove不存在的元素會報錯，discard不會。
* union, intersection: 快速獲得並集和交集，方便一些去重操作。

## dict

字典，相當於其它語言中的map, hashtable, hashmap之類的，讀取操作也是O(1) 複雜度

### keys(), values(), items()

這三個方法可以分別獲得key, value, {key: value}的陣列。

### setdefault

這個函數經常在初始化字典時候使用，如果某個key在字典中存在，返回它的value, 否則返回你給的 default 值。比如在建一個 trie 樹的時候

node = self.root

for char in word:

node = node.setdefault(char, {})

### OrderedDict

OrderedDict 能記錄你 key 和 value 插入的順序，底層其實是一個雙向鏈表加雜湊表的實現。我們甚至可以使用move\_to\_end這樣的函數：

>>> d = OrderedDict.fromkeys('abcde')

>>> d.move\_to\_end('b')

>>> ''.join(d.keys())

'acdeb'

# 放開頭

>>> d.move\_to\_end('b', last=False)

>>> ''.join(d.keys())

'bacde'

### defaultdict

defaultdict可以很好地來解決一些初始化的問題，比如 value 是一個 list，每次需要判斷 key 是否存在的情況。這時我們可以直接定義

d = defaultdict(list)

s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

for k, v in s:

d[k].append(v)

sorted(d.items())

# [('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

## heapq

heapq 就是 python 的 priority queue，heapq[0]即為堆頂元素。

heapq 的實現是小頂堆，如果需要一個大頂堆，常規的一個做法是把值取負存入，取出時再反轉。  
以下是借助 heapq 來實現 heapsort 的例子：

>>> def heapsort(iterable):

... h = []

... for value in iterable:

... heappush(h, value)

... return [heappop(h) for i in range(len(h))]

...

>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## bisect

python 自帶二分查找的庫，在一些不要求實現 binary search，但是借助它能加速的場景下可以直接使用。

bisect.bisect(a, x, lo=0, hi=len(a))

這裡的參數分別為 陣列，要查找的數，範圍起始點，範圍結束點

相似函數還有

* bisect.bisect\_left
* bisect.bisect\_right  
  分別返回可以插入 x 的最左和最右 index

## Counter

Counter 接受的參數可以是一個 string, 或者一個 list, mapping

>>> c = Counter() # a new, empty counter

>>> c = Counter('gallahad') # a new counter from an iterable

>>> c = Counter({'red': 4, 'blue': 2}) # a new counter from a mapping

>>> c = Counter(cats=4, dogs=8) # a new counter from keyword args

* most\_common(n)  
  可以得到出現次數最多的 n 個數：

>>> Counter('abracadabra').most\_common(3) # doctest: +SKIP

[('a', 5), ('r', 2), ('b', 2)]

## strings

### ord, char

ord 返回單個字元的 unicode:

>>> ord('a')

97

char 則是反向操作：

>>> chr(100)

'd'

### strip

移除 string 前後的字串，預設來移除空格，但是也可以給一個字串，然後會移除含有這個字串的部分：

>>> ' spacious '.strip()

'spacious'

>>> 'www.example.com'.strip('cmowz.')

'example'

### split

按照某個字串來切分，返回一個 list, 可以傳入一個參數maxsplit來限定分離數。

>>> '1,2,3'.split(',')

['1', '2', '3']

>>> '1,2,3'.split(',', maxsplit=1)

['1', '2,3']

>>> '1,2,,3,'.split(',')

['1', '2', '', '3', '']

## int/ float

### 最大, 最小 number

有時候初始化我們需要設定 Math.max() 和 Math.min(), 在 python 中分別以 float('inf') 和 float('-inf')表示

我們也可以這麼做：

import sys

#maxint

Max = sys.maxint

### 除法

在 python3 中， / 會保留浮點，相當於 float 相除，如果需要做到像 pyhton2 中的 int 相除，需要 //：

>>> 3 / 2

1.5

>>> 3 // 2

1

### 次方

在 python 中為 \*\*:

>>> 2 \*\* 10

1024

## conditions

在 python 的三項運算式(ternary operation) 與其它語言不太一樣：

res = a if condition else b

它表示如果 condition 滿足，那麼 res = a, 不然 res = b，在類 c 的語言裡即為：

res = condition ? a : b;

## any, all

any(), all()很好理解，就是字面意思，即參數中任何一個為 true 或者全部為 true 則返回 true。經常可以秀一些騷操作:  
比如 [36. Valid Sudoku](https://link.zhihu.com/?target=https%3A//leetcode.com/problems/valid-sudoku/" \t "_blank) 這題：

class Solution:

def isValidSudoku(self, board: List[List[str]]) -> bool:

row = [[x for x in y if x != '.'] for y in board]

col = [[x for x in y if x != '.'] for y in zip(\*board)]

pal = [[board[i+m][j+n] for m in range(3) for n in range(3) if board[i+m][j+n] != '.'] for i in (0, 3, 6) for j in (0, 3, 6)]

return all(len(set(x)) == len(x) for x in (\*row, \*col, \*pal))

## itertools

這是 python 自帶的反覆運算器庫，有很多實用的、與遍歷、反覆運算相關的函數。

### permutations 排列

permutations('ABCD', 2)

# AB AC AD BA BC BD CA CB CD DA DB DC

### combinations 組合

combinations('ABCD', 2)

# AB AC AD BC BD CD

### groupby 合併

[https://leetcode.com/problems/swap-for-longest-repeated-character-substring/discuss/355852/Python-Groupby/322898](https://link.zhihu.com/?target=https%3A//leetcode.com/problems/swap-for-longest-repeated-character-substring/discuss/355852/Python-Groupby/322898" \t "_blank)

[k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B

[list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D

## functools

這個庫裡有很多高階函數，包括前面介紹到的cmp\_to\_key 以及 reduce，但是比較逆天的有 lru\_cache，即 least recently used cache. 這個 [LRU Cache](https://link.zhihu.com/?target=https%3A//leetcode.com/problems/lru-cache/" \t "_blank)是一個常見的面試題，通常用 hashmap 和雙向鏈表來實現，python 居然直接內置了。

用法即直接作為 decorator 裝飾在要 cache 的函數上，以變數值為 key 存儲，當反復調用時直接返回計算過的值，例子如下：

### lru\_cache

[https://leetcode.com/problems/stone-game-ii/discuss/345230/Python-DP-Solution](https://link.zhihu.com/?target=https%3A//leetcode.com/problems/stone-game-ii/discuss/345230/Python-DP-Solution" \t "_blank)

def stoneGameII(self, A: List[int]) -> int:

N = len(A)

for i in range(N - 2, -1, -1):

A[i] += A[i + 1]

from functools import lru\_cache

@lru\_cache(None)

def dp(i, m):

if i + 2 \* m >= N: return A[i]

return A[i] - min(dp(i + x, max(m, x)) for x in range(1, 2 \* m + 1))

return dp(0, 1)

# **Python中的魔法函數**

# 介紹的魔法函數有（持續更新）： \_\_ init\_\_()、\_\_ str\_\_()、\_\_ new\_\_()、\_\_ unicode\_\_()、 \_\_ call\_\_()、 \_\_ len\_\_()、 \_\_repr\_\_()、\_\_ setattr\_\_()、 \_\_ getattr\_\_()、 \_\_ getattribute\_\_()、 \_\_ delattr\_\_()、\_\_ setitem\_\_()、 \_\_ getitem\_\_()、\_\_ delitem\_\_()、 \_\_ iter\_\_()、\_\_ del\_\_()、 \_\_dir\_\_()、\_\_dict\_\_()、\_\_exit\_\_()，\_\_enter(), \_\_all\_\_()等函數。

## **1. 前言**

### **1.1 什麼是魔法函數？**

所謂魔法函數（Magic Methods），是Python的一種高級語法，允許你在類中自定義函數（函數名格式一般為\_\_xx\_\_），並綁定到類的特殊方法中。比如在類A中自定義\_\_str\_\_()函數，則在調用str(A())時，會自動調用\_\_str\_\_()函數，並返回相應的結果。在我們平時的使用中，可能經常使用**\_\_init\_\_函數（構造函數）**和**\_\_del\_\_函數（析構函數）**，其實這也是魔法函數的一種。

* Python中以雙下劃線(\_\_xx\_\_)開始和結束的函數（不可自己定義）為魔法函數。
* 調用類實例化的對象的方法時自動調用魔法函數。
* 在自己定義的類中，可以實現之前的內置函數。

### **1.2 魔法函數有什麼作用？**

魔法函數可以為你寫的類增加一些額外功能，方便使用者理解。舉個簡單的例子，我們定義一個“人”的類People，當中有屬性姓名name、年齡age。讓你需要利用sorted函數對一個People的數組進行排序，排序規則是按照name和age同時排序，即name不同時比較name，相同時比較age。由於People類本身不具有比較功能，所以需要自定義，你可以這麼定義People類：

**class** **People**(object):

**def** **\_\_init\_\_**(self, name, age):

self**.**name **=** name

self**.**age **=** age

**return**

**def** **\_\_str\_\_**(self):

**return** self**.**name **+** ":" **+** str(self**.**age)

**def** **\_\_lt\_\_**(self, other):

**return** self**.**name **<** other**.**name **if** self**.**name **!=** other**.**name **else** self**.**age **<** other**.**age

**if** \_\_name\_\_**==**"\_\_main\_\_":

print("\t"**.**join([str(item) **for** item **in** sorted([People("abc", 18),

People("abe", 19), People("abe", 12), People("abc", 17)])]))

輸出結果：

abc:17 abc:18 abe:12 abe:19

上個例子中的\_\_lt\_\_函數即less than函數，即當比較兩個People實例時自動調用。

## **Python中有哪些魔法函數？**

Python中每個魔法函數都對應了一個Python內置函數或操作，比如\_\_str\_\_對應str函數，\_\_lt\_\_對應小於號<等。Python中的魔法函數可以大概分為以下幾類：

類的構造、刪除：

object**.**\_\_new\_\_(self, **...**)

object**.**\_\_init\_\_(self, **...**)

object**.**\_\_del\_\_(self)

二元操作符：

**+** object**.**\_\_add\_\_(self, other)

**-** object**.**\_\_sub\_\_(self, other)

**\*** object**.**\_\_mul\_\_(self, other)

**//** object**.**\_\_floordiv\_\_(self, other)

**/** object**.**\_\_div\_\_(self, other)

**%** object**.**\_\_mod\_\_(self, other)

**\*\*** object**.**\_\_pow\_\_(self, other[, modulo])

**<<** object**.**\_\_lshift\_\_(self, other)

**>>** object**.**\_\_rshift\_\_(self, other)

**&** object**.**\_\_and\_\_(self, other)

**^** object**.**\_\_xor\_\_(self, other)

**|** object**.**\_\_or\_\_(self, other)

擴展二元操作符：

**+=** object**.**\_\_iadd\_\_(self, other)

**-=** object**.**\_\_isub\_\_(self, other)

**\*=** object**.**\_\_imul\_\_(self, other)

**/=** object**.**\_\_idiv\_\_(self, other)

**//=** object**.**\_\_ifloordiv\_\_(self, other)

**%=** object**.**\_\_imod\_\_(self, other)

**\*\*=** object**.**\_\_ipow\_\_(self, other[, modulo])

**<<=** object**.**\_\_ilshift\_\_(self, other)

**>>=** object**.**\_\_irshift\_\_(self, other)

**&=** object**.**\_\_iand\_\_(self, other)

**^=** object**.**\_\_ixor\_\_(self, other)

**|=** object**.**\_\_ior\_\_(self, other)

一元操作符：

**-** object**.**\_\_neg\_\_(self)

**+** object**.**\_\_pos\_\_(self)

abs() object**.**\_\_abs\_\_(self)

**~** object**.**\_\_invert\_\_(self)

complex() object**.**\_\_complex\_\_(self)

int() object**.**\_\_int\_\_(self)

long() object**.**\_\_long\_\_(self)

float() object**.**\_\_float\_\_(self)

oct() object**.**\_\_oct\_\_(self)

hex() object**.**\_\_hex\_\_(self)

round() object**.**\_\_round\_\_(self, n)

floor() object\_\_floor\_\_(self)

ceil() object**.**\_\_ceil\_\_(self)

trunc() object**.**\_\_trunc\_\_(self)

比較函數：

**<** object**.**\_\_lt\_\_(self, other)

**<=** object**.**\_\_le\_\_(self, other)

**==** object**.**\_\_eq\_\_(self, other)

**!=** object**.**\_\_ne\_\_(self, other)

**>=** object**.**\_\_ge\_\_(self, other)

**>** object**.**\_\_gt\_\_(self, other)

類的表示、輸出：

str() object**.**\_\_str\_\_(self)

repr() object**.**\_\_repr\_\_(self)

len() object**.**\_\_len\_\_(self)

hash() object**.**\_\_hash\_\_(self)

bool() object**.**\_\_nonzero\_\_(self)

dir() object**.**\_\_dir\_\_(self)

sys**.**getsizeof() object**.**\_\_sizeof\_\_(self)

類容器：

len() object**.**\_\_len\_\_(self)

self[key] object**.**\_\_getitem\_\_(self, key)

self[key] **=** value object**.**\_\_setitem\_\_(self, key, value)

**del**[key] object**.**\_\_delitem\_\_(self, key)

iter() object**.**\_\_iter\_\_(self)

reversed() object**.**\_\_reversed\_\_(self)

in操作 object**.**\_\_contains\_\_(self, item)

字典key不存在時 object**.**\_\_missing\_\_(self, key)

## **2. 常見的魔法函數**

我們將魔法方法分為：非數學運算和數學運算兩大類。

### **2.1 非數學運算**

### **2.1.1 字串表示**

\_\_repr\_\_函數和\_\_str\_\_函數：

## **2.1.1.1. \_\_repr\_\_函數**

函數str() 用於將值轉化為適於人閱讀的形式，而repr() 轉化為供解釋器讀取的形式，某對象沒有適於人閱讀的解釋形式的話，str() 會返回與repr()，所以print展示的都是str的格式。

我們經常會直接輸出類的實例化對象，例如：

**class** **CLanguage**:

**pass**clangs **=** CLanguage()print(clangs)

程式運行結果為：

<\_\_main\_\_.CLanguage object at 0x000001A7275221D0>

通常情況下，直接輸出某個實例化對象，本意往往是想瞭解該對象的基本資訊，例如該對象有哪些屬性，它們的值各是多少等等。但默認情況下，我們得到的資訊只會是“類名+object at+記憶體地址”，對我們瞭解該實例化對象幫助不大。

那麼，有沒有可能自定義輸出實例化對象時的資訊呢？答案是肯定，通過重寫類的 \_\_repr\_\_() 方法即可。事實上，當我們輸出某個實例化對象時，其調用的就是該對象的 \_\_repr\_\_() 方法，輸出的是該方法的返回值。

以本節開頭的程式為例，執行 print(clangs) 等同於執行 print(clangs.\_\_repr\_\_())，程式的輸出結果是一樣的（輸出的記憶體地址可能不同）。

和 \_\_init\_\_(self) 的性質一樣，Python 中的每個類都包含 \_\_repr\_\_() 方法，因為 object 類包含 \_\_reper\_\_() 方法，而 Python 中所有的類都直接或間接繼承自 object 類。

默認情況下，\_\_repr\_\_() 會返回和調用者有關的 “類名+object at+記憶體地址”資訊。當然，我們還可以通過在類中重寫這個方法，從而實現當輸出實例化對象時，輸出我們想要的資訊。

舉個例子：

**class** **CLanguage**:

**def** **\_\_init\_\_**(self):

self**.**name **=** "C語言中文網"

self**.**add **=** "http://c.biancheng.net"

**def** **\_\_repr\_\_**(self):

**return** "CLanguage[name="**+** self**.**name **+**",add=" **+** self**.**add **+**"]"clangs **=** CLanguage()print(clangs)

程式運行結果為：

CLanguage[name**=**C語言中文網,add**=**http:**//**c**.**biancheng**.**net]

由此可見，\_\_repr\_\_() 方法是類的實例化對象用來做“自我介紹”的方法，默認情況下，它會返回當前對象的“類名+object at+記憶體地址”，而如果對該方法進行重寫，可以為其製作自定義的自我描述資訊。

## **2.1.1.2. \_\_ str\_\_()**

直接列印對象的實現方法，\_\_ str\_\_是被print函數調用的。列印一個實例化對象時，列印的其實時一個對象的地址。而通過\_\_str\_\_()函數就可以幫助我們列印對象中具體的屬性值，或者你想得到的東西。

在Python中調用print()列印實例化對象時會調用\_\_str\_\_()。如果\_\_str\_\_()中有返回值，就會列印其中的返回值。

**class** **Cat**:

"""定義一個貓類"""

**def** **\_\_init\_\_**(self, new\_name**=** "湯姆", new\_age**=** 20):

"""在創建完對象之後 會自動調用, 它完成對象的初始化的功能"""

self**.**name **=** new\_name

self**.**age **=** new\_age *# 它是一個對象中的屬性,在對象中存儲,即只要這個對象還存在,那麼這個變數就可以使用*

*# num = 100 # 它是一個局部變數,當這個函數執行完之後,這個變數的空間就沒有了,因此其他方法不能使用這個變數*

**def** **\_\_str\_\_**(self):

"""返回一個對象的描述資訊"""

*# print(num)*

**return** "名字是:%s , 年齡是:%d" **%** (self**.**name, self**.**age)

*# 創建了一個對象*tom **=** Cat("湯姆", 30)print(tom)

輸出結果：

名字是:湯姆 , 年齡是:30

總結：當使用print輸出對象的時候，只要自己定義了\_\_str\_\_(self)方法，那麼就會列印從在這個方法中return的數據。\_\_str\_\_方法需要返回一個字串，當做這個對象的描寫。

## **2.1.1.3. \_\_repr\_\_和\_\_str\_\_區別**

看下面的例子就明白了：

**class** **Test**(object):

**def** **\_\_init\_\_**(self, value**=**'hello, world!'):

self**.**data **=** value

**>>>** t **=** Test()**>>>** t**<**\_\_main\_\_**.**Test at 0x7fa91c307190**>>>>** print(t)**<**\_\_main\_\_**.**Test object at 0x7fa91c307190**>**

*# 看到了麼？上面列印類對象並不是很友好，顯示的是對象的記憶體地址# 下麵我們重構下該類的\_\_repr\_\_以及\_\_str\_\_，看看它們倆有啥區別*

*# 重構\_\_repr\_\_***class** **TestRepr**(Test):

**def** **\_\_repr\_\_**(self):

**return** 'TestRepr(%s)' **%** self**.**data

**>>>** tr **=** TestRepr()**>>>** tr 直接終端顯示，不print就是面向程式員TestRepr(hello, world!)**>>>** print(tr) print是面向程式員TestRepr(hello, world!)

*# 重構\_\_repr\_\_方法後，不管直接輸出對象還是通過print列印的資訊都按我們\_\_repr\_\_方法中定義的格式進行顯示了*

*# 重構\_\_str\_\_*calss TestStr(Test):

**def** **\_\_str\_\_**(self):

**return** '[Value: %s]' **%** self**.**data

**>>>** ts **=** TestStr()**>>>** ts**<**\_\_main\_\_**.**TestStr at 0x7fa91c314e50**>>>>** print(ts)[Value: hello, world!]

*# 你會發現，直接輸出對象ts時並沒有按我們\_\_str\_\_方法中定義的格式進行輸出，而用print輸出的資訊卻改變了*

\_\_repr\_\_和\_\_str\_\_這兩個方法都是用於顯示的，\_\_str\_\_是面向用戶的，而\_\_repr\_\_面向程式員。

* 列印操作會首先嘗試\_\_str\_\_和str內置函數(print運行的內部等價形式)，它通常應該返回一個友好的顯示。
* \_\_repr\_\_用於所有其他的環境中：用於交互模式下提示回應以及repr函數，如果沒有使用\_\_str\_\_，會使用print和str。它通常應該返回一個編碼字串，可以用來重新創建對象，或者給開發者詳細的顯示。

當我們想所有環境下都統一顯示的話，可以重構\_\_repr\_\_方法；當我們想在不同環境下支持不同的顯示，例如終端用戶顯示使用\_\_str\_\_，而程式員在開發期間則使用底層的\_\_repr\_\_來顯示，實際上\_\_str\_\_只是覆蓋了\_\_repr\_\_以得到更友好的用戶顯示。

### **2.1.2 集合、序列相關**

\_\_len\_\_函數、\_\_getitem\_\_函數、\_\_setitem\_\_函數、\_\_delitem\_\_函數和\_\_contains\_\_函數：

## **2.1.2.1. \_\_ len\_\_函數**

在Python中，如果你調用len()函數試圖獲取一個對象的長度，實際上，在len()函數內部，它自動去調用該對象的\_\_len\_\_()方法。

**class** **Students**():

**def** **\_\_init\_\_**(self, **\***args):

self**.**names **=** args

**def** **\_\_len\_\_**(self):

**return** len(self**.**names)

ss **=** Students('Bob', 'Alice', 'Tim')print(len(ss))

輸出結果：

3

## **2.1.2.2. \_\_getitem\_\_函數**

Python的特殊方法\_\_getitem\_() 主要作用是可以讓對象實現迭代功能。我們通過一個實例來說明。

定義一個Sentence類，通過索引提取單詞。

**import** reRE\_WORD **=** re**.**compile(r'\w+')**class** **Sentence**:

**def** **\_\_init\_\_**(self, text):

self**.**text **=** text

self**.**words **=** RE\_WORD**.**findall(text) *# re.findall函數返回一個字串列表，裏面的元素是正則運算式的全部非重疊匹配*

**def** **\_\_getitem\_\_**(self, index):

**return** self**.**words[index]

**測試：**

**>>>** s **=** Sentence('The time has come')**>>>** **for** word **in** s:

print(word)

The

time

has

come**>>>** s[0]

'The'**>>>** s[1]

'time'

通過測試發現，示例 s 可以正常迭代。但是沒有定義**getitem**() 測試則會報錯，TypeError: '\*\*\*' object is not iterable。

**序列可以迭代：**

我們都知道序列是可以迭代，下麵具體說明原因。

解釋器需要迭代對象x時， 會自動調用iter(x)方法。內置的 iter(x) 方法有以下作用：

* 檢查對象是否實現了\_\_iter\_\_ 方法，如果實現了就調用它（也就是我們偶爾用到的特殊方法重載），獲取一個迭代器。
* 如果沒有實現iter()方法， 但是實現了 \_\_getitem\_\_方法，Python會創建一個迭代器，嘗試按順序（從索引0開始，可以看到我們剛才是通過s[0]取值）獲取元素。
* 如果嘗試失敗，Python拋出TypeError異常，通常會提示TypeError: '\*\*\*' object is not iterable。

任何Python序列都可迭代的原因是，他們都實現了\_\_getitem\_\_方法。其實，標準的序列也都實現了\_\_iter\_\_方法。

**注意：**從python3.4 開始，檢查對象x能否迭代，最準確的方法是： 調用iter(x)方法，如果不可迭代，在處理TypeError異常。這比使用isinstance(x,abc.Iterable)更準確，因為iter()方法會考慮到遺留的\_\_getitem\_\_()方法，而abc.Iterable類則不考慮。

凡是在類中定義了這個\_\_getitem\_\_ 方法，那麼它的實例對象（假定為p），可以像這樣p[key] 取值，當實例對象做p[key] 運算時，會調用類中的方法\_\_getitem\_\_。

一般如果想使用索引訪問元素時，就可以在類中定義這個方法（\_\_getitem\_\_(self, key) ），當實例對象通過[] 運算符取值時，會調用它的方法\_\_getitem\_\_。

**class** **DataBase**:

'''Python 3 中的類'''

**def** **\_\_init\_\_**(self, id, address):

'''初始化方法'''

self**.**id **=** id

self**.**address **=** address

self**.**d **=** {self**.**id: 1,

self**.**address: "192.168.10.10",

}

**def** **\_\_getitem\_\_**(self, key):

*# return self.\_\_dict\_\_.get(key, "100")*

**return** self**.**d**.**get(key, "default")

data **=** DataBase(1, "192.168.2.11")

print(data["hi"])

print(data[data**.**id])

輸出結果：

default1

或者

**class** **DataBase**:

'''Python 3 中的類'''

**def** **\_\_init\_\_**(self, id, address):

'''初始化方法'''

self**.**id **=** id

self**.**address **=** address

**def** **\_\_getitem\_\_**(self, key):

**return** self**.**\_\_dict\_\_**.**get(key, "100")

data **=** DataBase(1, "192.168.2.11")

print(data["hi"])

print(data["id"])

輸出結果是：

1001

還可以用在對象的迭代上

**class** **STgetitem**:

**def** **\_\_init\_\_**(self, text):

self**.**text **=** text

**def** **\_\_getitem\_\_**(self, index):

result **=** self**.**text[index]**.**upper()

**return** result

p **=** STgetitem("Python")

print(p[0])

print("------------------------")

**for** char **in** p:

print(char)

輸出結果是：

P**------------------------**PYTHON

## **2.1.2.3. \_\_ setitem\_\_函數**

\_\_setitem\_\_(self,key,value)：該方法應該按一定的方式存儲和key相關的value。在設置類實例屬性時自動調用的。

*# -\*- coding:utf-8 -\*-*

**class** **A**:

**def** **\_\_init\_\_**(self):

self['B']**=**'BB'

self['D']**=**'DD'

**def** **\_\_setitem\_\_**(self,name,value):

print "\_\_setitem\_\_:Set %s Value %s" **%**(name,value)

**if** \_\_name\_\_**==**'\_\_main\_\_':

X**=**A()

輸出結果為：

\_\_setitem\_\_:Set B Value BB\_\_setitem\_\_:Set D Value DD

## **2.1.2.4. \_\_ delitem\_\_()**

\_\_delitem\_\_(self,key):

這個方法在對對象的組成部分使用\_\_del\_\_語句的時候被調用，應刪除與key相關聯的值。同樣，僅當對象可變的時候，才需要實現這個方法。

**class** **Tag**:

**def** **\_\_init\_\_**(self):

self**.**change**=**{'python':'This is python',

'php':'PHP is a good language'}

**def** **\_\_getitem\_\_**(self, item):

print('調用getitem')

**return** self**.**change[item]

**def** **\_\_setitem\_\_**(self, key, value):

print('調用setitem')

self**.**change[key]**=**value

**def** **\_\_delitem\_\_**(self, key):

print('調用delitem')

**del** self**.**change[key]

a**=**Tag()

print(a['php'])

**del** a['php']

print(a**.**change)

輸出結果：

調用getitemPHP **is** a good language調用delitem{'python': 'This is python'}

## **2.1.2.5. \_\_contains\_\_函數**

在Class裏添加\_\_contains\_\_(self,x)函數,可判斷我們輸入的數據是否在Class裏.參數x就是我們傳入的數據.

如下代碼:

**class** **Graph**():

**def** **\_\_init\_\_**(self):

self**.**items **=** {'a':1,'b':2,'c':3}

**def** **\_\_contains\_\_**(self,x): *# 判斷一個定點是否包含在裏面*

**return** x **in** self**.**items

a **=** Graph()

print('a' **in** a) *# 返回True*

print('d' **in** a) *# 返回False*

**>>** **True>>** **False**

**class** **Graph**():

**def** **\_\_init\_\_**(self):

self**.**items **=** {'a':1,'b':2,'c':3}

**def** **\_\_str\_\_**(self):

**return** '列印我幹嘛'

**def** **\_\_contains\_\_**(self,x): *# x參數接受的就是我們手動傳遞的數據*

**if** x**<**10 **and** x**>**0:

**return** **True**

**return** **False**

print(9 **in** Graph())

print(5 **in** Graph())

print(51 **in** Graph())

**>>** **True>>** **True>>** **False**

### **2.1.3 迭代相關**

\_\_iter\_\_函數和\_\_next\_\_函數：

## **\_\_ iter\_\_函數和\_\_next\_\_函數**

迭代器就是重複地做一些事情，可以簡單的理解為迴圈，在python中實現了\_\_iter\_\_方法的對象是可迭代的，實現了next()方法的對象是迭代器，這樣說起來有點拗口，實際上要想讓一個迭代器工作，至少要實現\_\_iter\_\_方法和next方法。很多時候使用迭代器完成的工作使用列表也可以完成，但是如果有很多值列表就會佔用太多的記憶體，而且使用迭代器也讓我們的程式更加通用、優雅、pythonic。

如果一個類想被用於for ... in迴圈，類似list或tuple那樣，就必須實現一個\_\_iter\_\_()方法，該方法返回一個迭代對象，然後，Python的for迴圈就會不斷調用該迭代對象的next()方法拿到迴圈的下一個值，直到遇到StopIteration錯誤時退出迴圈。

**容器（container）：**

容器是用來儲存元素的一種數據結構，容器將所有數據保存在內存中，Python中典型的容器有：list，set，dict，str等等。

**class** **test**():

**def** **\_\_init\_\_**(self,data**=**1):

self**.**data **=** data

**def** **\_\_iter\_\_**(self):

**return** self

**def** **\_\_next\_\_**(self):

**if** self**.**data **>** 5:

**raise** **StopIteration**

**else**:

self**.**data**+=**1

**return** self**.**data

**for** item **in** test(3):

print(item)

輸出結果：

456

for … in… 這個語句其實做了兩件事。第一件事是獲得一個可迭代器，即調用了\_\_iter\_\_()函數。 第二件事是迴圈的過程，迴圈調用\_\_next\_\_()函數。

對於test這個類來說，它定義了\_\_iter\_\_和\_\_next\_\_函數，所以是一個可迭代的類，也可以說是一個可迭代的對象（Python中一切皆對象）。

**迭代器：**

含有\_\_next\_\_()函數的對象都是一個迭代器，所以test也可以說是一個迭代器。如果去掉\_\_itet\_\_()函數，test這個類也不會報錯。如下代碼所示：

**class** **test**():

**def** **\_\_init\_\_**(self,data**=**1):

self**.**data **=** data

**def** **\_\_next\_\_**(self):

**if** self**.**data **>** 5:

**raise** **StopIteration**

**else**:

self**.**data**+=**1

**return** self**.**data

t **=** test(3) **for** i **in** range(3):

print(t**.**\_\_next\_\_())

輸出結果：

456

**生成器：**

生成器是一種特殊的迭代器。當調用fib()函數時，生成器實例化並返回，這時並不會執行任何代碼，生成器處於空閒狀態，注意這裏prev, curr = 0, 1並未執行。然後這個生成器被包含在list()中，list會根據傳進來的參數生成一個列表，所以它對fib()對象(一切皆對象，函數也是對象)調用\_\_next\_\_方法。

**def** **fib**(end **=** 1000):

prev,curr**=**0,1

**while** curr **<** end:

**yield** curr

prev,curr**=**curr,curr**+**prev

print(list(fib()))

輸出結果：

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]

上面只是做了幾個演示，這裏具體說明一下:

當調用iter函數的時候，生成了一個迭代對象，要求\_\_iter\_\_必須返回一個實現了\_\_next\_\_的對象，我們就可以通過next函數訪問這個對象的下一個元素了，並且在你不想繼續有迭代的情況下拋出一個StopIteration的異常(for語句會捕獲這個異常，並且自動結束for)，下麵實現了一個自己的類似range函數的功能。

**class** **MyRange**(object):

**def** **\_\_init\_\_**(self, end):

self**.**start **=** 0

self**.**end **=** end

**def** **\_\_iter\_\_**(self):

**return** self

**def** **\_\_next\_\_**(self):

**if** self**.**start **<** self**.**end:

ret **=** self**.**start

self**.**start **+=** 1

**return** ret

**else**:

**raise** **StopIteration**

**from** collections.abc **import** **\***

a **=** MyRange(5)

print(isinstance(a, Iterable))

print(isinstance(a, Iterator))

**for** i **in** a:

print(i)

輸出結果是：

**TrueTrue**01234

接下來我們使用next函數模擬一次：

**class** **MyRange**(object):

**def** **\_\_init\_\_**(self, end):

self**.**start **=** 0

self**.**end **=** end

**def** **\_\_iter\_\_**(self):

**return** self

**def** **\_\_next\_\_**(self):

**if** self**.**start **<** self**.**end:

ret **=** self**.**start

self**.**start **+=** 1

**return** ret

**else**:

**raise** **StopIteration**

a **=** MyRange(5)

print(next(a))

print(next(a))

print(next(a))

print(next(a))

print(next(a))

print(next(a)) *# 其實到這裏已經完成了，我們在運行一次查看異常*

可以看見一個很明顯的好處是，每次產生的數據，是產生一個用一個，什麼意思呢，比如我要遍曆[0, 1, 2, 3.....]一直到10億，如果使用列表的方式，那麼是會全部載入記憶體的，但是如果使用迭代器，可以看見，當用到了(也就是在調用了next)才會產生對應的數字，這樣就可以節約記憶體了，這是一種懶惰的加載方式。

## **總結**

1. 可以使用collection.abs裏面的Iterator和Iterable配合isinstance函數來判斷一個對象是否是可迭代的，是否是迭代器對象。
2. iter實際是映射到了\_\_iter\_\_函數。
3. 只要實現了\_\_iter\_\_的對象就是可迭代對象(Iterable)，正常情況下，應該返回一個實現了\_\_next\_\_的對象(雖然這個要求不強制)，如果自己實現了\_\_next\_\_，當然也可以返回自己。
4. 同時實現了\_\_iter\_\_和\_\_next\_\_的是迭代器(Iterator)，當然也是一個可迭代對象了，其中\_\_next\_\_應該在迭代完成後，拋出一個StopIteration異常。
5. for語句會自動處理這個StopIteration異常以便結束for迴圈。

### **2.1.4 可調用**

\_\_call\_\_函數：

### **\_\_ call\_\_函數**

該方法的功能類似於在類中重載 () 運算符，使得類實例對象可以像調用普通函數那樣，以“對象名()”的形式使用。**作用：**為了將**類的實例對象**變為可調用對象。

**class** **CLanguage**:

*# 定義\_\_call\_\_方法*

**def** **\_\_call\_\_**(self,name,add):

print("調用\_\_call\_\_()方法",name,add)

clangs **=** CLanguage()clangs("C語言中文網","http://c.biancheng.net")

程式執行結果為：

調用\_\_call\_\_()方法 C語言中文網 http:**//**c**.**biancheng**.**net

可以看到，通過在 CLanguage 類中實現 \_\_call\_\_() 方法，使的 clangs 實例對象變為了可調用對象。

Python 中，凡是可以將 () 直接應用到自身並執行，都稱為可調用對象。可調用對象包括自定義的函數、Python 內置函數以及本節所講的類實例對象。

對於可調用對象，實際上“名稱()”可以理解為是“名稱.\_\_call\_\_()”的簡寫。仍以上面程式中定義的 clangs 實例對象為例，其最後一行代碼還可以改寫為如下形式：

clangs**.**\_\_call\_\_("C語言中文網","http://c.biancheng.net")

運行程式會發現，其運行結果和之前完全相同。

用 \_\_call\_\_() 彌補 hasattr() 函數的短板：hasattr() 函數的用法，該函數的功能是查找類的實例對象中是否包含指定名稱的屬性或者方法，但該函數有一個缺陷，即它無法判斷該指定的名稱，到底是類屬性還是類方法。

要解決這個問題，我們可以借助可調用對象的概念。要知道，類實例對象包含的方法，其實也屬於可調用對象，但類屬性卻不是。舉個例子：

**class** **CLanguage**:

**def** **\_\_init\_\_** (self):

self**.**name **=** "C語言中文網"

self**.**add **=** "http://c.biancheng.net"

**def** **say**(self):

print("我正在學Python")clangs **=** CLanguage()**if** hasattr(clangs,"name"):

print(hasattr(clangs**.**name,"\_\_call\_\_"))print("\*\*\*\*\*\*\*\*\*\*")**if** hasattr(clangs,"say"):

print(hasattr(clangs**.**say,"\_\_call\_\_"))

程式執行結果為：

**False\*\*\*\*\*\*\*\*\*\*True**

可以看到，由於 name 是類屬性，它沒有以 \_\_call\_\_ 為名的 \_\_call\_\_() 方法；而 say 是類方法，它是可調用對象，因此它有 \_\_call\_\_() 方法。

### **其他實例**

1. 函數本身可以被調用：

def func():

pass

print(callable(func))

輸出結果：

**True**

2. 類本身可以被調用，主要用作生成實例化對象：

**class** **Fun**:

**def** **\_\_init\_\_**(self):

**pass**

print(callable(Fun))

輸出結果：

**True**

3. 類的實例化對象無法被調用：

**class** **Fun**:

**def** **\_\_init\_\_**(self):

**pass**

a **=** Fun()

print(callable(a))

輸出結果：

**False**

4. 通過增加\_\_call\_\_()函數，將類實例化對象變為可調用：

**class** **Fun**:

**def** **\_\_init\_\_**(self):

**pass**

**def** **\_\_call\_\_**(self, **\***args, **\*\***kwargs):

**pass**

a **=** Fun()print(callable(a))

輸出結果：

**True**

### **2.1.5 with上下文管理器**

\_\_enter\_\_函數和\_\_exit\_\_函數：

### **\_\_exit\_\_函數，\_\_enter\_\_函數**

\_\_exit\_\_和\_\_enter\_\_函數是與with語句的組合應用的，用於上下文管理。

**1.\_\_enter(self)\_\_：**

負責返回一個值，該返回值將賦值給as子句後面的var\_name，通常返回對象自己，即“self”。函數優先於with後面的“代碼塊”(statements1,statements2,……)被執行。

**2.\_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb)：**

**with** xxx **as** var\_name：

*# 代碼塊開始*

statements1

statements2

……

*# 代碼塊結束*

*# 代碼快後面的語句*statements after code block

執行完with後面的代碼塊後自動調用該函數。with語句後面的“代碼塊”中有異常(不包括因調用某函數,由被調用函數內部拋出的異常) ，會把異常類型，異常值，異常跟蹤資訊分別賦值給函數參數exc\_type, exc\_val, exc\_tb，沒有異常的情況下，exc\_type, exc\_val, exc\_tb值都為None。另外，如果該函數返回True、1類值的Boolean真值，那麼將忽略“代碼塊”中的異常，停止執行“代碼塊”中剩餘語句，但是會繼續執行“代碼塊”後面的語句；如果函數返回類似0，False類的Boolean假值、或者沒返回值，將拋出“代碼塊”中的異常，那麼在沒有捕獲異常的情況下，中斷“代碼塊”及“代碼塊”之後語句的執行。

**代碼：**

**class** **User**(object):

**def** **\_\_init\_\_**(self, username, password):

self**.**\_username **=** username

self**.**\_password **=** password

@property

**def** **username**(self):

**return** self**.**\_username

@username**.**setter

**def** **username**(self, username):

self**.**\_username **=** username

@property

**def** **password**(self):

**return** self**.**\_password

@password**.**setter

**def** **password**(self, password):

self**.**\_password **=** password

**def** **\_\_enter\_\_**(self):

print('before：auto do something before statements body of with executed')

**return** self

**def** **\_\_exit\_\_**(self, exc\_type, exc\_val, exc\_tb):

print('after：auto do something after statements body of with executed')

**if** \_\_name\_\_ **==** '\_\_main\_\_':

boy **=** User('faker', 'faker2021')

print(boy**.**password)

print("上下文管理器with語句：")

**with** User('faker', 'faker2021') **as** user:

print(user**.**password)

print('---------end-----------')

輸出結果：

faker2021

上下文管理器with語句：

before：auto do something before statements body of **with** executed

faker2021

after：auto do something after statements body of **with** executed

**---------**end**-----------**

**更改上述部分代碼如下，繼續運行：**

**class** **User**(object):

**def** **\_\_init\_\_**(self, username, password):

self**.**\_username **=** username

self**.**\_password **=** password

@property

**def** **username**(self):

**return** self**.**\_username

@username**.**setter

**def** **username**(self, username):

self**.**\_username **=** username

@property

**def** **password**(self):

**return** self**.**\_password

@password**.**setter

**def** **password**(self, password):

self**.**\_password **=** password

**def** **\_\_enter\_\_**(self):

print('before：auto do something before statements body of with executed')

**return** self

**def** **\_\_exit\_\_**(self, exc\_type, exc\_val, exc\_tb):

print('auto do something after statements body of with executed')

print('exc\_type:', exc\_type)

print('exc\_val:', exc\_val)

print('exc\_tb:', exc\_tb)

**return** **False**

**if** \_\_name\_\_ **==** '\_\_main\_\_':

boy **=** User('faker', 'faker2021')

print(boy**.**password)

print("上下文管理器with語句：")

**with** User('faker', 'faker2021') **as** user:

print(user**.**password)

12**/**0

print('after execption')

print('---------end-----------')

輸出結果：

Traceback (most recent call last):

faker2021

File "/code/0.py", line 42, **in** **<**module**>**

上下文管理器with語句：

12**/**0

before：auto do something before statements body of **with** executed

**ZeroDivisionError**: division by zero

faker2021

auto do something after statements body of **with** executed

exc\_type: **<class** '**ZeroDivisionError**'>

exc\_val: division by zero

exc\_tb: **<**traceback object at 0x0000015F4A62AD48**>**

**在上述的基礎上繼續修改代碼，將\_\_exit\_\_的返回值設置為True：**

**def** **\_\_exit\_\_**(self, exc\_type, exc\_val, exc\_tb):

print('auto do something after statements body of with executed')

print('exc\_type:', exc\_type)

print('exc\_val:', exc\_val)

print('exc\_tb:', exc\_tb)

**return** **True**

輸出結果：

faker2021

上下文管理器with語句：

before：auto do something before statements body of **with** executed

faker2021

auto do something after statements body of **with** executed

exc\_type: **<class** '**ZeroDivisionError**'>

exc\_val: division by zero

exc\_tb: **<**traceback object at 0x0000021DBDD3AD48**>**

**---------**end**-----------**

注意：

1、拋異常後，代碼塊中剩餘的語句沒有再繼續運行

2、如果在上述的基礎上，把代碼中的 12/0剪切後放到password(self)中 ，拋出異常的異常資訊也會傳遞給\_\_exit\_\_函數的。

@property**def** **password**(self):

12**/**0

**return** self**.**\_password

**if** \_\_name\_\_ **==** '\_\_main\_\_':

print("上下文管理器with語句：")

**with** User('faker', 'faker2021') **as** user:

print(user**.**password)

print('---------end-----------')

輸出結果：

上下文管理器with語句：

before：auto do something before statements body of **with** executed

auto do something after statements body of **with** executed

exc\_type: **<class** '**ZeroDivisionError**'>

exc\_val: division by zero

exc\_tb: **<**traceback object at 0x000001614FFFAF88**>**

**---------**end**-----------**

### **2.1.6 數值轉換**

\_\_abs\_\_函數、\_\_bool\_\_函數、\_\_int\_\_函數、\_\_float\_\_函數、\_\_hash\_\_函數和\_\_index\_\_函數：

### **2.1.7 元類相關**

\_\_new\_\_函數和\_\_init\_\_函數：

## **2.1.7.1. \_\_ new\_\_函數**

\_\_new\_\_() 是一種負責創建類實例的靜態方法，它無需使用 staticmethod 裝飾器修飾，且該方法會優先 \_\_init\_\_() 初始化方法被調用。

一般情況下，覆寫 \_\_new\_\_() 的實現將會使用合適的參數調用其超類的 super().\_\_new\_\_()，並在返回之前修改實例。例如：

**class** **demoClass**:

instances\_created **=** 0

**def** **\_\_new\_\_**(cls,**\***args,**\*\***kwargs):

print("\_\_new\_\_():",cls,args,kwargs)

instance **=** super()**.**\_\_new\_\_(cls)

instance**.**number **=** cls**.**instances\_created

cls**.**instances\_created **+=** 1

**return** instance

**def** **\_\_init\_\_**(self,attribute):

print("\_\_init\_\_():",self,attribute)

self**.**attribute **=** attributetest1 **=** demoClass("abc")test2 **=** demoClass("xyz")print(test1**.**number,test1**.**instances\_created)print(test2**.**number,test2**.**instances\_created)

輸出結果：

\_\_new\_\_(): **<class** '**\_\_main\_\_.**demoClass'> ('abc',) {}\_\_init\_\_(): **<**\_\_main\_\_**.**demoClass object at 0x0000025650FACF28**>** abc\_\_new\_\_(): **<class** '**\_\_main\_\_.**demoClass'> ('xyz',) {}\_\_init\_\_(): **<**\_\_main\_\_**.**demoClass object at 0x000002565FFC4CF8**>** xyz0 21 2

\_\_new\_\_() 通常會返回該類的一個實例，但有時也可能會返回其他類的實例，如果發生了這種情況，則會跳過對 \_\_init\_\_() 方法的調用。而在某些情況下（比如需要修改不可變類實例（Python 的某些內置類型）的創建行為），利用這一點會事半功倍。比如：

**class** **nonZero**(int):

**def** **\_\_new\_\_**(cls,value):

**return** super()**.**\_\_new\_\_(cls,value) **if** value **!=** 0 **else** **None**

**def** **\_\_init\_\_**(self,skipped\_value):

*#此例中會跳過此方法*

print("\_\_init\_\_()")

super()**.**\_\_init\_\_()

print(type(nonZero(**-**12)))

print(type(nonZero(0)))

輸出結果：

\_\_init\_\_()**<class** '**\_\_main\_\_.**nonZero'>**<class** '**NoneType**'>

那麼，什麼情況下使用 \_\_new\_\_() 呢？答案很簡單，在 \_\_init\_\_() 不夠用的時候。

例如，前面例子中對 Python 不可變的內置類型（如 int、str、float 等）進行了子類化，這是因為一旦創建了這樣不可變的對象實例，就無法在 \_\_init\_\_() 方法中對其進行修改。

有些讀者可能會認為，\_\_new\_\_() 對執行重要的對象初始化很有用，如果用戶忘記使用 super()，可能會漏掉這一初始化。雖然這聽上去很合理，但有一個主要的缺點，即如果使用這樣的方法，那麼即便初始化過程已經是預期的行為，程式員明確跳過初始化步驟也會變得更加困難。不僅如此，它還破壞了“\_\_init\_\_() 中執行所有初始化工作”的潛規則。

注意，由於 \_\_new\_\_() 不限於返回同一個類的實例，所以很容易被濫用，不負責任地使用這種方法可能會對代碼有害，所以要謹慎使用。一般來說，對於特定問題，最好搜索其他可用的解決方案，最好不要影響對象的創建過程，使其違背程式員的預期。比如說，前面提到的覆寫不可變類型初始化的例子，完全可以用工廠方法（一種設計模式）來替代。

## **2.1.7.2. \_\_ init\_\_()**

**所有類的超類object**，有一個默認包含pass的\_\_ init \_\_()實現，這個函數會在對象初始化的時候調用，我們可以選擇實現，也可以選擇不實現，一般建議是實現的，不實現對象屬性就不會被初始化。

**\_\_init\_\_() 方法可以包含多個參數，但必須包含一個名為 self 的參數，且必須作為第一個參數。**也就是說，類的構造方法最少也要有一個 self 參數，僅包含 self 參數的 \_\_init\_\_() 構造方法，又稱為類的默認構造方法。例如，仍以 TheFirstDemo 類為例，添加構造方法的代碼如下所示：

**class** **TheFirstDemo**:

'''這是一個學習Python定義的第一個類'''

*# 構造方法*

**def** **\_\_init\_\_**(self):

print("調用構造方法")

*# 下麵定義了一個類屬性*

add **=** 'http://c.biancheng.net'

*# 下麵定義了一個say方法*

**def** **say**(self, content):

print(content)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

result **=** TheFirstDemo()

輸出結果：

調用構造方法

在創建 result 這個對象時，隱式調用了我們手動創建的 \_\_init\_\_() 構造方法。

不僅如此，在 \_\_init\_\_() 構造方法中，除了 self 參數外，還可以自定義一些參數，參數之間使用逗號“,”進行分割。例如，下麵的代碼在創建 \_\_init\_\_() 方法時，額外指定了 2 個參數：

**class** **CLanguage**:

'''這是一個學習Python定義的一個類'''

**def** **\_\_init\_\_**(self,name,add):

print(name,"的網址為:",add)

*#創建 add 對象，並傳遞參數給構造函數*add **=** CLanguage("C語言中文網","http://c.biancheng.net")

輸出結果：

C語言中文網 的網址為: http:**//**c**.**biancheng**.**net

可以看到，雖然構造方法中有 self、name、add 3 個參數，但實際需要傳參的僅有 name 和 add，也就是說，self 不需要手動傳遞參數。

### **2.1.8 屬性相關**

\_\_getattr\_\_函數、\_\_setattr\_\_函數、\_\_getattribute\_\_函數、\_\_setattribute\_\_函數和\_\_dir\_\_函數：

### **2.1.8.1. \_\_ getattr\_\_函數**

當我們訪問一個不存在的屬性的時候，會拋出異常，提示我們不存在這個屬性。而這個異常就是\_\_getattr\_\_方法拋出的，其原因在於他是訪問一個不存在的屬性的最後落腳點，作為異常拋出的地方提示出錯再適合不過了。

看例子，我們找一個存在的屬性和不存在的屬性：

**class** **A**(object):

**def** **\_\_init\_\_**(self, value):

self**.**value **=** value

**def** **\_\_getattr\_\_**(self, item):

print("into \_\_getattr\_\_")

**return** "can not find"

a **=** A(10)

print(a**.**value)*# 10*

print(a**.**name)*# into \_\_getattr\_\_# can not find*

輸出結果：

10

into \_\_getattr\_\_can **not** find

### **2.1.8.2. \_\_ setattr\_\_函數**

在類中對屬性進行賦值操作時，python會自動調用\_\_setattr\_\_()函數，來實現對屬性的賦值。但是重寫\_\_setattr\_\_()函數時要注意防止無限遞歸的情況出現，一般解決辦法有兩種，一是用通過super()調用\_\_setatrr\_\_()函數，二是利用字典操作對相應鍵直接賦值。

簡單的說，\_\_setattr\_\_()在屬性賦值時被調用，並且將值存儲到實例字典中，這個字典應該是self的\_\_dict\_\_屬性。即：**在類實例的每個屬性進行賦值時，都會首先調用\_\_setattr\_\_()方法，並在\_\_setattr\_\_()方法中將屬性名和屬性值添加到類實例的\_\_dict\_\_屬性中**。

**實例屬性管理\_\_dict\_\_：**

下麵的測試代碼中定義了三個實例屬性，每個實例屬性註冊後都print()此時的\_\_dict\_\_，代碼如下：

**class** **AnotherFun**:

**def** **\_\_init\_\_**(self):

self**.**name **=** "Liu"

print(self**.**\_\_dict\_\_)

self**.**age **=** 12

print(self**.**\_\_dict\_\_)

self**.**male **=** **True**

print(self**.**\_\_dict\_\_)another\_fun **=** AnotherFun()

得到的結果顯示出，每次實例屬性賦值時，都會將屬性名和對應值存儲到\_\_dict\_\_字典中：

{'name': 'Liu'}{'name': 'Liu', 'age': 12}{'name': 'Liu', 'age': 12, 'male': **True**}

**\_\_setattr\_\_()與\_\_dict\_\_：**

由於每次類實例進行屬性賦值時都會調用\_\_setattr\_\_()，所以可以重載\_\_setattr\_\_()方法，來動態的觀察每次實例屬性賦值時\_\_dict\_\_()的變化。下麵的Fun類重載了\_\_setattr\_\_()方法，並且將實例的屬性和屬性值作為\_\_dict\_\_的鍵-值對：

**class** **Fun**:

**def** **\_\_init\_\_**(self):

self**.**name **=** "Liu"

self**.**age **=** 12

self**.**male **=** **True**

**def** **\_\_setattr\_\_**(self, key, value):

print("\*"**\***50)

print("setting:{}, with:{}"**.**format(key, value))

print("current \_\_dict\_\_ : {}"**.**format(self**.**\_\_dict\_\_))

*# 屬性註冊*

self**.**\_\_dict\_\_[key] **=** valuefun **=** Fun()

通過在\_\_setattr\_\_()中將屬性名作為key，並將屬性值作為value，添加到了\_\_dict\_\_中，得到的結果如下：

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***setting:name, **with**:Liucurrent \_\_dict\_\_ : {}**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***setting:age, **with**:12current \_\_dict\_\_ : {'name': 'Liu'}**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***setting:male, **with**:**True**current \_\_dict\_\_ : {'name': 'Liu', 'age': 12}

可以看出，\_\_init\_\_()中三個屬性賦值時，每次都會調用一次\_\_setattr\_\_()函數。

**重載\_\_setattr\_\_()必須謹慎：**

由於\_\_setattr\_\_()負責在\_\_dict\_\_中對屬性進行註冊，所以自己在重載時必須進行屬性註冊過程，下麵是\_\_setattr\_\_()不進行屬性註冊的例子：

**class** **NotFun**:

**def** **\_\_init\_\_**(self):

self**.**name **=** "Liu"

self**.**age **=** 12

self**.**male **=** **True**

**def** **\_\_setattr\_\_**(self, key, value):

**pass**not\_fun **=** NotFun()print(not\_fun**.**name)

由於\_\_setattr\_\_中並沒有將屬性註冊到\_\_dict\_\_中，所以not\_fun對象並沒有name屬性，因此最後的print（not\_fun.name）會報出屬性不存在的錯誤：

**AttributeError** Traceback (most recent call last)**<**ipython**-**input**-**21**-**6158d7aaef71**>** **in** **<**module**>**()

8 **pass**

9 not\_fun **=** NotFun()**--->** 10 print(not\_fun**.**name)

**AttributeError**: 'NotFun' object has no attribute 'name'

所以，重載\_\_setattr\_\_時必須要考慮是否在\_\_dict\_\_中進行屬性註冊。

**總結：**Python的實例屬性的定義、獲取和管理可以通過\_\_setattr\_\_()和\_\_dict\_\_配合進行，當然還有對應的\_\_getattr\_\_()方法，如上文所示。\_\_setattr\_\_()方法在類的屬性賦值時被調用，並通常需要把屬性名和屬性值存儲到self的\_\_dict\_\_字典中。

### **2.1.8.3. \_\_ getattribute\_\_函數**

首先理解\_\_getattribute\_\_的用法，先看代碼：

**class** **Tree**(object):

**def** **\_\_init\_\_**(self,name):

self**.**name **=** name

self**.**cate **=** "plant"

**def** **\_\_getattribute\_\_**(self,obj):

print("哈哈")

**return** object**.**\_\_getattribute\_\_(self,obj)

aa **=** Tree("大樹")

print(aa**.**name)

執行結果是：

哈哈大樹

為什麼會這個結果呢？

\_\_getattribute\_\_是屬性訪問攔截器，就是當這個類的屬性被訪問時，會自動調用類的\_\_getattribute\_\_方法。即在上面代碼中，當我調用實例對象aa的name屬性時，不會直接列印，而是把name的值作為實參傳進\_\_getattribute\_\_方法中（參數obj是我隨便定義的，可任意起名），經過一系列操作後，再把name的值返回。Python中只要定義了繼承object的類，就默認存在屬性攔截器，只不過是攔截後沒有進行任何操作，而是直接返回。所以我們可以自己改寫\_\_getattribute\_\_方法來實現相關功能，比如查看許可權、列印log日誌等。如下代碼，簡單理解即可：

**class** **Tree**(object):

**def** **\_\_init\_\_**(self,name):

self**.**name **=** name

self**.**cate **=** "plant"

**def** **\_\_getattribute\_\_**(self,**\***args,**\*\***kwargs):

**if** args[0] **==** "大樹"

print("log 大樹")

**return** "我愛大樹"

**else**:

**return** object**.**\_\_getattribute\_\_(self,**\***args,**\*\***kwargs)aa **=** Tree("大樹")print(aa**.**name)print(aa**.**cate)

結果是：

log 大樹我愛大樹plant

**另外，注意注意：**

初學者用\_\_getattribute\_\_方法時，容易栽進這個坑，什麼坑呢，直接看代碼：

**class** **Tree**(object):

**def** **\_\_init\_\_**(self,name):

self**.**name **=** name

self**.**cate **=** "plant"

**def** **\_\_getattribute\_\_**(self,obj):

**if** obj**.**endswith("e"):

**return** object**.**\_\_getattribute\_\_(self,obj)

**else**:

**return** self**.**call\_wind()

**def** **call\_wind**(self):

**return** "樹大招風"aa **=** Tree("大樹")

print(aa**.**name)*#因為name是以e結尾，所以返回的還是name，所以列印出"大樹"*

print(aa**.**wind)*#這個代碼中因為wind不是以e結尾，#所以返回self.call\_wind的結果,列印的是"樹大招風"*

**上面的解釋正確嗎？**

先說結果，關於print(aa.name)的解釋是正確的，但關於print(aa.wind)的解釋不對，為什麼呢？我們來分析一下，執行aa.wind時，先調用\_\_getattribute\_\_方法，經過判斷後，它返回的是self.call\_wind()，即self.call\_wind的執行結果，但當去調用aa這個對象的call\_wind屬性時，前提是又要去調用\_\_getattribute\_\_方法，反反復複，沒完沒了，形成了遞歸調用且沒有退出機制，最終程式就掛了！

### **2.1.8.4. \_\_dir\_\_函數**

dir() 函數，通過此函數可以某個對象擁有的所有的屬性名和方法名，該函數會返回一個包含有所有屬性名和方法名的有序列表。

舉個例子：

**class** **CLanguage**:

**def** **\_\_init\_\_** (self,):

self**.**name **=** "C語言中文網"

self**.**add **=** "http://c.biancheng.net"

**def** **say**():

**pass**clangs **=** CLanguage()

print(dir(clangs))

程式運行結果為：

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'add', 'name', 'say']

注意，通過 dir() 函數，不僅僅輸出本類中新添加的屬性名和方法（最後 3 個），還會輸出從父類（這裏為 object 類）繼承得到的屬性名和方法名。

值得一提的是，dir() 函數的內部實現，其實是在調用參數對象 \_\_dir\_\_() 方法的基礎上，對該方法返回的屬性名和方法名做了排序。

所以，除了使用 dir() 函數，我們完全可以自行調用該對象具有的 \_\_dir\_\_() 方法：

**class** **CLanguage**:

**def** **\_\_init\_\_** (self,):

self**.**name **=** "C語言中文網"

self**.**add **=** "http://c.biancheng.net"

**def** **say**():

**pass**clangs **=** CLanguage()

print(clangs**.**\_\_dir\_\_())

程式運行結果為：

['name', 'add', '\_\_module\_\_', '\_\_init\_\_', 'say', '\_\_dict\_\_', '\_\_weakref\_\_', '\_\_doc\_\_', '\_\_repr\_\_', '\_\_hash\_\_', '\_\_str\_\_', '\_\_getattribute\_\_', '\_\_setattr\_\_', '\_\_delattr\_\_', '\_\_lt\_\_', '\_\_le\_\_', '\_\_eq\_\_', '\_\_ne\_\_', '\_\_gt\_\_', '\_\_ge\_\_', '\_\_new\_\_', '\_\_reduce\_ex\_\_', '\_\_reduce\_\_', '\_\_subclasshook\_\_', '\_\_init\_subclass\_\_', '\_\_format\_\_', '\_\_sizeof\_\_', '\_\_dir\_\_', '\_\_class\_\_']

顯然，使用 \_\_dir\_\_() 方法和 dir() 函數輸出的數據是相同，僅僅順序不同。

### **2.1.9 屬性描述符**

\_\_get\_\_函數、\_\_set\_\_函數和\_\_delete\_函數：

## **\_get\_\_函數、\_\_set\_\_函數和\_\_delete\_函數**

\_\_get\_\_函數、\_\_set\_\_函數和\_\_delete\_函數是描述符。

一般來說,描述符是帶有“綁定行為”的對象屬性,它的屬性訪問已經被描述符協議中的方法覆蓋了.這些方法是\_\_get\_\_(),\_\_set\_\_(),和\_\_delete\_\_()。如果一個對象定義了這些方法中的任何一個,它就是一個描述符。

默認的屬相訪問是從對象的字典中 get, set, 或者 delete 屬性,；例如a.x的查找順序是: a.x -> a.\_\_dict\_\_['x'] -> type(a).\_\_dict\_\_['x'] -> type(a)的基類(不包括元類),如果查找的值是對象定義的描述方法之一,python可能會調用描述符方法來重載默認行為,發生在這個查找環節的哪里取決於定義了哪些描述符方法。

注意,只有在新式類中描述符才會起作用(新式類繼承type或者object class)。描述符是強有力的通用協議,屬性、方法、靜態方法、類方法和super()背後使用的就是這個機制,描述符簡化了底層的c代碼,並為Python編程提供了一組靈活的新工具

### **2.1.9.1. 那描述符是什麼？**

描述符本質就是一個新式類,在這個新式類中,至少實現了\_\_get\_\_(),\_\_set\_\_(),\_\_delete\_\_()中的一個,這也被稱為描述符協議。

\_\_get\_\_():調用一個屬性時,觸發

\_\_set\_\_():為一個屬性賦值時,觸發

\_\_delete\_\_():採用del刪除屬性時,觸發

**class** **Foo**: *#在python3中Foo是新式類,它實現了三種方法,這個類就被稱作一個描述符*

**def** **\_\_get\_\_**(self, instance, owner):

**pass**

**def** **\_\_set\_\_**(self, instance, value):

**pass**

**def** **\_\_delete\_\_**(self, instance):

**pass**

### **2.1.9.2. 描述符是幹什麼的？**

描述符的作用是用來代理另外一個類的屬性的(必須把描述符定義成這個類的類屬性，不能定義到構造函數中)。

**class** **Foo**:

**def** **\_\_get\_\_**(self, instance, owner):

print('觸發get')

**def** **\_\_set\_\_**(self, instance, value):

print('觸發set')

**def** **\_\_delete\_\_**(self, instance):

print('觸發delete')

*#包含這三個方法的新式類稱為描述符,由這個類產生的實例進行屬性的調用/賦值/刪除,並不會觸發這三個方法*f1**=**Foo()f1**.**name**=**'egon'f1**.**name**del** f1**.**name

*#描述符Str# 該描述符的作用是為另外一個類的類屬性進行服務,另外一個類就是描述符的對象***class** **Str**:

**def** **\_\_get\_\_**(self, instance, owner):

print('Str調用')

print(instance, owner)

**def** **\_\_set\_\_**(self, instance, value):

print('Str設置...')

print(instance,value)

**def** **\_\_delete\_\_**(self, instance):

print('Str刪除...')

print(instance)

**class** **People**:

name **=** Str() *# name 屬性被代理,將這個類作用於另外一個類的屬性來使用*

**def** **\_\_init\_\_**(self,name): *#name被Str類代理*

self**.**name **=** name

p1 **=** People('long')

*# 描述符Str的使用*p1**.**namep1**.**name **=** 'sss'**del** p1**.**name

print(p1**.**\_\_dict\_\_)

print(People**.**\_\_dict\_\_)

'''

結果：

Str設置...

<\_\_main\_\_.People object at 0x000001E6CA585C88> long

Str調用

<\_\_main\_\_.People object at 0x000001E6CA585C88> <class '\_\_main\_\_.People'>

Str設置...

<\_\_main\_\_.People object at 0x000001E6CA585C88> sss

Str刪除...

<\_\_main\_\_.People object at 0x000001E6CA585C88>

{}

{'\_\_module\_\_': '\_\_main\_\_', 'name': <\_\_main\_\_.Str object at 0x000001E6CA585BE0>, '\_\_init\_\_': <function People.\_\_init\_\_ at 0x000001E6CA584048>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'People' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'People' objects>, '\_\_doc\_\_': None}

'''

### **2.1.9.3. 描述符的種類：兩種。**

### **2.1.9.3.1 數據描述符:至少實現了\_\_get\_\_()和\_\_set\_\_()**

**class** **Foo**:

**def** **\_\_set\_\_**(self, instance, value):

print('set')

**def** **\_\_get\_\_**(self, instance, owner):

print('get')

### **2.1.9.3.2 非數據描述符:沒有實現\_\_set\_\_()**

class Foo:

def \_\_get\_\_(self, instance, owner):

print('get')

### **2.1.9.4. 注意事項**

1. 描述符本身應該定義成新式類,被代理的類也應該是新式類。
2. 必須把描述符定義成這個類的類屬性，不能為定義到構造函數中。
3. 要嚴格遵循該優先順序,優先順序由高到底分別是。

* 1.類屬性
* 2.數據描述符
* 3.實例屬性
* 4.非數據描述符
* 5.找不到的屬性觸發\_\_getattr\_\_()

### **2.1.9.5. 描述符使用**

眾所周知，python是弱類型語言，即參數的賦值沒有類型限制，下麵我們通過描述符機制來實現類型限制功能。

*#描述符***class** **Typed**:

**def** **\_\_init\_\_**(self,key):

self**.**key **=** key

**def** **\_\_get\_\_**(self, instance, owner):

print('get方法')

*# print('instance參數：%s'%instance) # People object*

*# print('owner參數：%s'%owner)*

**return** instance**.**\_\_dict\_\_[self**.**key]

**def** **\_\_set\_\_**(self, instance, value):

*# 代理的好處:可以對傳進來的值進行下一步判斷*

print('set方法')

*# print('instance參數：%s'%instance) # People Object*

*# print('value參數：%s'%value)*

**if** **not** isinstance(value,str):

**raise** **TypeError**('你傳入的值不是字元類型')

instance**.**\_\_dict\_\_[self**.**key] **=** value

**def** **\_\_delete\_\_**(self, instance):

print('delete方法')

*# print('instance參數：%s'%instance) # People object*

instance**.**\_\_dict\_\_**.**pop(self**.**key)

**class** **People**:

name **=** Typed('name') *# 代理類*

**def** **\_\_init\_\_**(self,name,age,salary):

self**.**name **=** name

self**.**age **=** age

self**.**salary **=** salary

p1 **=** People('asd',18,330) *# (數據描述符>實例)創建實例時，觸發\_\_set\_\_*

print(p1**.**\_\_dict\_\_)

print('-----------------')

p1**.**name **=** 'log'

print(p1**.**\_\_dict\_\_)

print('------------------')

name **=** p1**.**name

print(name)

print('------------------')

**del** p1**.**name

print(p1**.**\_\_dict\_\_)

'''

結果：

set方法

{'name': 'asd', 'age': 18, 'salary': 330}

set方法

{'name': 'log', 'age': 18, 'salary': 330}

get方法

Log

delete方法

{'age': 18, 'salary': 330}

'''

*# -------------如果傳入的name不是字串類型-----------------*

p2 **=** People(321,18,330)

'''Traceback (most recent call last):

set方法

File "F:/Python3/老男孩/面向對象/day4/描述符應用.py", line 60, in <module>

p2 = People(321,18,330)

File "F:/Python3/老男孩/面向對象/day4/描述符應用.py", line 29, in \_\_init\_\_

self.name = name

File "F:/Python3/老男孩/面向對象/day4/描述符應用.py", line 18, in \_\_set\_\_

raise TypeError('你傳入的值不是字元類型')

TypeError: 你傳入的值不是字元類型

'''

*#描述符***class** **Typed**:

**def** **\_\_init\_\_**(self,key,key\_type):

self**.**key **=** key

self**.**key\_type **=** key\_type

**def** **\_\_get\_\_**(self, instance, owner):

print('get方法')

*# print('instance參數：%s'%instance) # People object*

*# print('owner參數：%s'%owner)*

**return** instance**.**\_\_dict\_\_[self**.**key]

**def** **\_\_set\_\_**(self, instance, value):

*# 代理的好處:可以對傳進來的值進行下一步判斷*

print('set方法')

*# print('instance參數：%s'%instance) # People Object*

*# print('value參數：%s'%value)*

**if** **not** isinstance(value,self**.**key\_type):

**raise** **TypeError**('%s不是%s'**%**(value,self**.**key\_type))

instance**.**\_\_dict\_\_[self**.**key] **=** value

**def** **\_\_delete\_\_**(self, instance):

print('delete方法')

*# print('instance參數：%s'%instance) # People object*

instance**.**\_\_dict\_\_**.**pop(self**.**key)

**class** **People**:

name **=** Typed('name',str) *# 代理類*

age **=** Typed('age',int)

**def** **\_\_init\_\_**(self,name,age,salary):

self**.**name **=** name

self**.**age **=** age

self**.**salary **=** salary

*# ----------------------3 多個值進行判斷-------------------*

p1 **=** People('long','18',2000)

'''

Traceback (most recent call last):

set方法

File "F:/Python3/老男孩/面向對象/day4/描述符應用.py", line 76, in <module>

set方法

p1 = People('long','18',2000)

File "F:/Python3/老男孩/面向對象/day4/描述符應用.py", line 32, in \_\_init\_\_

self.age = age

File "F:/Python3/老男孩/面向對象/day4/描述符應用.py", line 18, in \_\_set\_\_

raise TypeError('%s不是%s'%(value,self.key\_type))

TypeError: 18不是<class 'int'>

'''

### **2.1.10 協程**

\_\_await\_\_函數、\_\_aiter\_\_函數、\_\_anext\_\_函數、\_\_aenter\_\_函數和\_\_aexit\_\_函數

### **2.2 數學運算**

### **2.2.1 一元運算符**

\_\_neg\_\_ (-)、\_\_pos\_\_ (+)和\_\_abs\_\_函數。

### **2.2.2 二元運算符**

\_\_lt\_\_ (<)、\_\_le\_\_ (<=)、\_\_eq\_\_ (==)、\_\_ne\_\_ (!=)、\_\_gt\_\_ (>)和\_\_ge\_\_ (>=)。

### **2.2.3 算術運算符**

\_\_add\_\_ (+)、\_\_sub\_\_ (-)、\_\_mul\_\_ (\*)、\_\_truediv\_\_ (/)、\_\_floordiv\_\_ (//)、\_\_mod\_\_ (%)、\_\_divmod\_\_ 或divmod()、\_\_pow\_\_ 或pow() (\*\*)和\_\_round\_\_ 或round()。

### **2.2.4 反向算術運算符**

\_\_radd\_\_、\_\_rsub\_\_、\_\_rmul\_\_、\_\_rtruediv\_\_、\_\_rfloordiv\_\_、\_\_rmod\_\_、\_\_rdivmod\_\_和\_\_rpow\_\_。

### **2.2.5 增量賦值算術運算符**

\_\_iadd\_\_、\_\_isub\_\_、\_\_imul\_\_、\_\_ifloordiv\_\_和\_\_ipow\_\_。

### **2.2.6 位運算符**

\_\_invert\_\_ (~)、\_\_lshift\_\_ (<<)、\_\_rshift\_\_ (>>)、\_\_and\_\_ (&)、\_\_or\_\_ (|)和\_\_xor\_\_ (^)。

### **2.2.7 反向位運算符**

\_\_rlshift\_\_、\_\_rrshift\_\_、\_\_iand\_\_、\_\_ixor\_\_和\_\_ior\_\_。

### **2.2.8 增量賦值運算符**

\_\_ilshift\_\_、\_\_irshift\_\_、\_\_iand\_\_、\_\_ixor\_\_和\_\_ior\_\_。

### **2.3 其他魔法函數**

\_\_ unicode\_\_()函數，\_\_ delattr\_\_()函數， \_\_ del\_\_()函數， \_\_dict\_\_()函數，\_\_all\_\_()函數：

### **2.3.1. \_\_ unicode\_\_()**

\_\_ unicode\_\_()方法是在一個對象上調用unicode()時被調用的。因為Django的資料庫後端會返回Unicode字串給model屬性，所以我們通常會給自己的model寫一個\_\_ unicode\_\_()方法。如果定義了\_\_ unicode\_\_()方法但是沒有定義\_\_ str\_\_()方法，Django會自動提供一個\_\_ str\_\_()方法調用 \_\_ unicode\_\_()方法，然後把結果轉換為UTF-8編碼的字串對象，所以在一般情況下，只定義\_\_ unicode\_\_()方法，讓 Django來處理字串對象的轉換，看一個小栗子：

**class** **Demo**(object):

**def** **\_\_init\_\_**(self):

self**.**a **=** 1

**def** **\_\_unicode\_\_**(self):

**return** f"the value is {self.a}"

print(unicode(Demo()))

輸出結果：

the value **is** 1

在django中，雖然沒有定義\_\_ str\_\_，但是django會將\_\_ unicode\_\_轉為了str，當然你調用unicode更加是沒有問題的。

### **2.3.2. \_\_ delattr\_\_()**

本函數的作用是刪除屬性，實現了該函數的類可以用del 命令來刪除屬性。

**class** **MyClass**:

**def** **\_\_init\_\_**(self, work, score):

self**.**work **=** work

self**.**score **=** score

**def** **\_\_delattr\_\_**(self, name):

print("你正在刪除一個屬性")

**return** super()**.**\_\_delattr\_\_(name)

**def** **main**():

test **=** MyClass(work**=**"math", score**=**100)

*# 刪除work屬性*

**del** test**.**work

*# work屬性刪除,score屬性還在*

print(test**.**score)

**try**:

print(test**.**work)

**except** **AttributeError** **as** reason:

print(reason)

**if** \_\_name\_\_ **==** '\_\_main\_\_':

main()

輸出結果：

你正在刪除一個屬性100'MyClass' object has no attribute 'work'

### **2.3.3. \_\_ del\_\_()**

\_\_del\_\_() 方法，功能正好和 \_\_init\_\_() 相反，其用來銷毀實例化對象。

事實上在編寫程式時，如果之前創建的類實例化對象後續不再使用，最好在適當位置手動將其銷毀，釋放其佔用的記憶體空間（整個過程稱為垃圾回收（簡稱GC））。

大多數情況下，Python 開發者不需要手動進行垃圾回收，因為 Python 有自動的垃圾回收機制（下麵會講），能自動將不需要使用的實例對象進行銷毀。

無論是手動銷毀，還是 Python 自動幫我們銷毀，都會調用 \_\_del\_\_() 方法。舉個例子：

**class** **CLanguage**:

**def** **\_\_init\_\_**(self):

print("調用 \_\_init\_\_() 方法構造對象")

**def** **\_\_del\_\_**(self):

print("調用\_\_del\_\_() 銷毀對象，釋放其空間")clangs **=** CLanguage()**del** clangs

程式運行結果為：

調用 \_\_init\_\_() 方法構造對象調用\_\_del\_\_() 銷毀對象，釋放其空間

但是，讀者千萬不要誤認為，只要為該實例對象調用 \_\_del\_\_() 方法，該對象所佔用的記憶體空間就會被釋放。舉個例子：

**class** **CLanguage**:

**def** **\_\_init\_\_**(self):

print("調用 \_\_init\_\_() 方法構造對象")

**def** **\_\_del\_\_**(self):

print("調用\_\_del\_\_() 銷毀對象，釋放其空間")clangs **=** CLanguage()*#添加一個引用clangs對象的實例對象*cl **=** clangs**del** clangsprint("\*\*\*\*\*\*\*\*\*\*\*")

程式運行結果為：

調用 \_\_init\_\_() 方法構造對象**\*\*\*\*\*\*\*\*\*\*\***調用\_\_del\_\_() 銷毀對象，釋放其空間

注意，最後一行輸出資訊，是程式執行即將結束時調用 \_\_del\_\_() 方法輸出的。

可以看到，當程式中有其他變數（比如這裏的 cl）引用該實例對象時，即便手動調用 \_\_del\_\_() 方法，該方法也不會立即執行。這和 Python 的垃圾回收機制的實現有關。

Python 採用自動引用計數（簡稱 ARC）的方式實現垃圾回收機制。該方法的核心思想是：每個 Python 對象都會配置一個計數器，初始 Python 實例對象的計數器值都為 0，如果有變數引用該實例對象，其計數器的值會加 1，依次類推；反之，每當一個變數取消對該實例對象的引用，計數器會減 1。如果一個 Python 對象的的計數器值為 0，則表明沒有變數引用該 Python 對象，即證明程式不再需要它，此時 Python 就會自動調用 \_\_del\_\_() 方法將其回收。

以上面程式中的 clangs 為例，實際上構建 clangs 實例對象的過程分為 2 步，先使用 CLanguage() 調用該類中的 \_\_init\_\_() 方法構造出一個該類的對象（將其稱為 C，計數器為 0），並立即用 clangs 這個變數作為所建實例對象的引用（ C 的計數器值 + 1）。在此基礎上，又有一個 clang 變數引用 clangs（其實相當於引用 CLanguage()，此時 C 的計數器再 +1 ），這時如果調用del clangs語句，只會導致 C 的計數器減 1（值變為 1），因為 C 的計數器值不為 0，因此 C 不會被銷毀（不會執行 \_\_del\_\_() 方法）。

如果在上面程式結尾，添加如下語句：

**del** clprint("-----------")

則程式的執行結果為：

調用 \_\_init\_\_() 方法構造對象**\*\*\*\*\*\*\*\*\*\*\***調用\_\_del\_\_() 銷毀對象，釋放其空間**-----------**

可以看到，當執行 del cl 語句時，其應用的對象實例對象 C 的計數器繼續 -1（變為 0），對於計數器為 0 的實例對象，Python 會自動將其視為垃圾進行回收。

需要額外說明的是，如果我們重寫子類的 \_\_del\_\_() 方法（父類為非 object 的類），則必須顯式調用父類的 \_\_del\_\_() 方法，這樣才能保證在回收子類對象時，其佔用的資源（可能包含繼承自父類的部分資源）能被徹底釋放。為了說明這一點，這裏舉一個反例：

**class** **CLanguage**:

**def** **\_\_del\_\_**(self):

print("調用父類 \_\_del\_\_() 方法")**class** **cl**(CLanguage):

**def** **\_\_del\_\_**(self):

print("調用子類 \_\_del\_\_() 方法")c **=** cl()**del** c

程式運行結果為：

調用子類 \_\_del\_\_() 方法

### **2.3.4. \_\_dict\_\_()**

在 Python 類的內部，無論是類屬性還是實例屬性，都是以字典的形式進行存儲的，其中屬性名作為鍵，而值作為該鍵對應的值。

為了方便用戶查看類中包含哪些屬性，Python 類提供了\_\_dict\_\_ 屬性。需要注意的一點是，該屬性可以用類名或者類的實例對象來調用，用類名直接調用 \_\_dict\_\_，會輸出該由類中所有類屬性組成的字典；而使用類的實例對象調用 \_\_dict\_\_，會輸出由類中所有實例屬性組成的字典。

舉個例子：

**class** **CLanguage**:

a **=** 1

b **=** 2

**def** **\_\_init\_\_** (self):

self**.**name **=** "C語言中文網"

self**.**add **=** "http://c.biancheng.net"*#通過類名調用\_\_dict\_\_*print(CLanguage**.**\_\_dict\_\_)*#通過類實例對象調用 \_\_dict\_\_*clangs **=** CLanguage()print(clangs**.**\_\_dict\_\_)

程式輸出結果為：

{'\_\_module\_\_': '\_\_main\_\_', 'a': 1, 'b': 2, '\_\_init\_\_': **<**function CLanguage**.**\_\_init\_\_ at 0x0000022C69833E18**>**, '\_\_dict\_\_': **<**attribute '\_\_dict\_\_' of 'CLanguage' objects**>**, '\_\_weakref\_\_': **<**attribute '\_\_weakref\_\_' of 'CLanguage' objects**>**, '\_\_doc\_\_': **None**}{'name': 'C語言中文網', 'add': 'http://c.biancheng.net'}

不僅如此，對於具有繼承關係的父類和子類來說，父類有自己的 \_\_dict\_\_，同樣子類也有自己的 \_\_dict\_\_，它不會包含父類的 \_\_dict\_\_。例如：

**class** **CLanguage**:

a **=** 1

b **=** 2

**def** **\_\_init\_\_** (self):

self**.**name **=** "C語言中文網"

self**.**add **=** "http://c.biancheng.net"

**class** **CL**(CLanguage):

c **=** 1

d **=** 2

**def** **\_\_init\_\_** (self):

self**.**na **=** "Python教程"

self**.**ad **=** "http://c.biancheng.net/python"*#父類名調用\_\_dict\_\_*print(CLanguage**.**\_\_dict\_\_)*#子類名調用\_\_dict\_\_*print(CL**.**\_\_dict\_\_)*#父類實例對象調用 \_\_dict\_\_*clangs **=** CLanguage()print(clangs**.**\_\_dict\_\_)*#子類實例對象調用 \_\_dict\_\_*cl **=** CL()print(cl**.**\_\_dict\_\_)

運行結果為：

{'\_\_module\_\_': '\_\_main\_\_', 'a': 1, 'b': 2, '\_\_init\_\_': **<**function CLanguage**.**\_\_init\_\_ at 0x000001721A853E18**>**, '\_\_dict\_\_': **<**attribute '\_\_dict\_\_' of 'CLanguage' objects**>**, '\_\_weakref\_\_': **<**attribute '\_\_weakref\_\_' of 'CLanguage' objects**>**, '\_\_doc\_\_': **None**}{'\_\_module\_\_': '\_\_main\_\_', 'c': 1, 'd': 2, '\_\_init\_\_': **<**function CL**.**\_\_init\_\_ at 0x000001721CD15510**>**, '\_\_doc\_\_': **None**}{'name': 'C語言中文網', 'add': 'http://c.biancheng.net'}{'na': 'Python教程', 'ad': 'http://c.biancheng.net/python'}

顯然，通過子類直接調用的 \_\_dict\_\_ 中，並沒有包含父類中的 a 和 b 類屬性；同樣，通過子類對象調用的 \_\_dict\_\_，也沒有包含父類對象擁有的 name 和 add 實例屬性。

除此之外，借助由類實例對象調用 \_\_dict\_\_ 屬性獲取的字典，可以使用字典的方式對其中實例屬性的值進行修改，例如：

**class** **CLanguage**:

a **=** "aaa"

b **=** 2

**def** **\_\_init\_\_** (self):

self**.**name **=** "C語言中文網"

self**.**add **=** "http://c.biancheng.net"*#通過類實例對象調用 \_\_dict\_\_*clangs **=** CLanguage()print(clangs**.**\_\_dict\_\_)clangs**.**\_\_dict\_\_['name'] **=** "Python教程"print(clangs**.**name)

程式運行結果為：

{'name': 'C語言中文網', 'add': 'http://c.biancheng.net'}Python教程

注意，無法通過類似的方式修改類變數的值。

### **2.3.5. \_\_all\_\_()**

python模組中的\_\_all\_\_，用於模組導入時限制，如：from module import \*

此時被導入模組若定義了\_\_all\_\_屬性，則只有\_\_all\_\_內指定的屬性、方法、類可被導入；若沒定義，則導入模組內的所有公有屬性，方法和類。

**實例1：**

bb.py：

**class** **A**():

**def** **\_\_init\_\_**(self,name,age):

self**.**name**=**name

self**.**age**=**age**class** **B**():

**def** **\_\_init\_\_**(self,name,id):

self**.**name**=**name

self**.**id**=**id**def** **fun**():

print("func() is run!")**def** **fun1**():

print("func1() is run!")

test\_bb.py：

**from** bb **import** **\***a**=**A('zhansan','18')print(a**.**name,a**.**age)b**=**B("lisi",1001)print(b**.**name,b**.**id)fun()fun1()

運行結果：由於bb.py中沒有定義\_\_all\_\_屬性，所以導入了bb.py中所有的公有屬性

zhansan 18lisi 1001func() **is** run!func1() **is** run!

**實例2 :**

bb.py：

\_\_all\_\_**=**('A','func')**class** **A**():

**def** **\_\_init\_\_**(self,name,age):

self**.**name**=**name

self**.**age**=**age**class** **B**():

**def** **\_\_init\_\_**(self,name,id):

self**.**name**=**name

self**.**id**=**id**def** **func**():

print("func() is run!")**def** **func1**():

print("func1() is run!")

test\_bb.py :

**from** bb **import** **\***a**=**A('zhansan','18')print(a**.**name,a**.**age)func()*#b=B("lisi",1001)#NameError: name 'B' is not defined#func1()#NameError: name 'func1' is not defined*

運行結果：由於bb.py中使用了\_\_all\_\_=('A','func')，所以在別的模組導入該模組時，只能導入\_\_all\_\_中的變數、方法、類

zhansan 18func() **is** run!

**實例3：**

bb.py :

**def** **func**(): *#模組中的public方法*

print('func() is run!')**def** **\_func**(): *#模組中的protected方法*

print('\_func() is run!')**def** **\_\_func**(): *#模組中的private方法*

print('\_\_func() is run!')

test\_bb.py :

**from** bb **import** **\*** *#此方式只能導入公有的屬性、方法、類【無法導入以單下劃線開頭（protected）或以雙下劃線開頭(private)的屬性、方法、類】*func()*#\_func()#\_\_func()*

運行結果：

1. from bb import \* --此方式只能導入公有的屬性、方法、類【無法導入以單下劃線開頭protected或以雙下劃線開頭private的屬性、方法、類】
2. \_func() #NameError: name '\_func' is not defined
3. \_\_func() #NameError: name '\_\_func' is not defined

func() **is** run!

**實例4：**

bb.py :

\_\_all\_\_**=**('func','\_\_func','\_A')*#放入\_\_all\_\_中所有屬性均可導入，即使是以下劃線開頭***class** **\_A**():

**def** **\_\_init\_\_**(self,name):

self**.**name**=**name**def** **func**():

print("func() is run!")**def** **func1**():

print("func1() is run!")**def** **\_func**():

print("\_func() is run!")**def** **\_\_func**():

print("\_\_func() is run!")

test\_bb.py :

**from** bb **import** **\***func()*#func1()#func1不在\_\_all\_\_中，無法導入 NameError: name 'func1' is not defined #\_func()#\_func不在\_\_all\_\_中，無法導入 NameError: name '\_func' is not defined*\_\_func()*#\_\_func在\_\_all\_\_中，可以導入*a**=**\_A('zhangsan')*#\_A在\_\_all\_\_中，可以導入*print(a**.**name)

運行結果：

1. 放入\_\_all\_\_中所有屬性均可導入，即使是以下劃線開頭
2. func1() #func1不在\_\_all\_\_中，無法導入 NameError: name 'func1' is not defined
3. \_func() #\_func不在\_\_all\_\_中，無法導入 NameError: name '\_func' is not defined
4. \_\_func() #\_\_func在\_\_all\_\_中，可以導入
5. a=\_A('python') #\_A在\_\_all\_\_中，可以導入

func() **is** run!\_\_func() **is** run!zhangsan

**實例5:**

bb.py:

**def** **func**():

print('func() is run!')**def** **\_func**():

print('\_func() is run!')**def** **\_\_func**():

print('\_\_func() is run!')

test\_bb.py :

**from** bb **import** func,\_func,\_\_func *#可以通過這種方式導入public,protected,private*func()\_func()\_\_func()

運行結果：雖然\_func()、\_\_func()屬於protected , private許可權的，但是如果使用該方式，是可以直接導入訪問的

func() **is** run!\_func() **is** run!\_\_func() **is** run!

**實例6:**

bb.py :

**def** **func**():

print('func() is run!')**def** **\_func**():

print('\_func() is run!')**def** **\_\_func**():

print('\_\_func() is run!')

test\_bb.py :

**import** bb *#可以通過這種方式導入public,protected,private*bb**.**func()bb**.**\_func()bb**.**\_\_func()

運行結果：可以通過import模組的方式導入模組，然後使用模組.XX的方式訪問public,protected,private許可權的內容

func() **is** run!\_func() **is** run!\_\_func() **is** run!

# **Python 魔法方法指南**

## **01. 構造方法**

我們最為熟知的基本的魔法方法就是 \_\_init\_\_ ，我們可以用它來指明一個對象初始化的行為。然而，當我們調用 x = SomeClass() 的時候， \_\_init\_\_ 並不是第一個被調用的方法。事實上，第一個被調用的是 \_\_new\_\_ ，這個 方法才真正地創建了實例。當這個對象的生命週期結束的時候， \_\_del\_\_ 會被調用。讓我們近一步理解這三個方法：

* \_\_new\_\_(cls,[…)  
  \_\_new\_\_ 是對象實例化時第一個調用的方法，它只取下 cls 參數，並把其他參數傳給 \_\_init\_\_ 。\_\_new\_\_ 很少使用，但是也有它適合的場景，尤其是當類繼承自一個像元組或者字串這樣不經常改變的類型的時候。我不打算深入討論 \_\_new\_\_ ，因為它並不是很有用， Python文檔 中 有詳細的說明。
* \_\_init\_\_(self,[…])  
  類的初始化方法。它獲取任何傳給構造器的參數（比如我們調用 x = SomeClass(10, 'foo') ， \_\_init\_\_ 就會接到參數 10 和 'foo' 。\_\_init\_\_ 在Python的類定義中用的最多。
* \_\_del\_\_(self)  
  \_\_new\_\_ 和 \_\_init\_\_ 是對象的構造器， \_\_del\_\_ 是對象的銷毀器。它並非實現了語句 del x (因此該語句不等同於 x.\_\_del\_\_())。而是定義了當對象被垃圾回收時的行為。當對象需要在銷毀時做一些處理的時候這個方法很有用，比如 socket 對象、檔對象。但是需要注意的是，當Python解釋器退出但對象仍然存活的時候， \_\_del\_\_ 並不會 執行。所以養成一個手工清理的好習慣是很重要的，比如及時關閉連接。

這裏有個 \_\_init\_\_ 和 \_\_del\_\_ 的例子:

**from** os.path **import** join

**class** **FileObject**:

'''檔對象的裝飾類，用來保證檔被刪除時能夠正確關閉。'''

**def** **\_\_init\_\_**(self, filepath**=**'~', filename**=**'sample.txt'):

*# 使用讀寫模式打開filepath中的filename檔*

self**.**file **=** open(join(filepath, filename), 'r+')

**def** **\_\_del\_\_**(self):

self**.**file**.**close()

**del** self**.**file

## **02. 操作符**

使用Python魔法方法的一個巨大優勢就是可以構建一個擁有Python內置類型行為的對象。這意味著你可以避免使用非標準的、醜陋的方式來表達簡單的操作。在一些語言中，這樣做很常見:

**if** instance**.**equals(other\_instance):

*# do something*

你當然可以在Python也這麼做，但是這樣做讓代碼變得冗長而混亂。不同的類庫可能對同一種比較操作採用不同的方法名稱，這讓使用者需要做很多沒有必要的工作。運用魔法方法的魔力，我們可以定義方法 \_\_eq\_\_

**if** instance **==** other\_instance:

*# do something*

這是魔法力量的一部分，這樣我們就可以創建一個像內建類型那樣的對象了！

### **2.1 比較操作符**

Python包含了一系列的魔法方法，用於實現對象之間直接比較，而不需要採用方法調用。同樣也可以重載Python默認的比較方法，改變它們的行為。下麵是這些方法的列表：

* \_\_cmp\_\_(self, other)  
  \_\_cmp\_\_ 是所有比較魔法方法中最基礎的一個，它實際上定義了所有比較操作符的行為（<,==,!=,等等），但是它可能不能按照你需要的方式工作（例如，判斷一個實例和另一個實例是否相等採用一套標準，而與判斷一個實例是否大於另一實例採用另一套）。\_\_cmp\_\_ 應該在 self < other 時返回一個負整數，在 self == other 時返回0，在 self > other 時返回正整數。最好只定義你所需要的比較形式，而不是一次定義全部。如果你需要實現所有的比較形式，而且它們的判斷標準類似，那麼 \_\_cmp\_\_ 是一個很好的方法，可以減少代碼重複，讓代碼更簡潔。
* \_\_eq\_\_(self, other)  
  定義等於操作符(==)的行為。
* \_\_ne\_\_(self, other)  
  定義不等於操作符(!=)的行為。
* \_\_lt\_\_(self, other)  
  定義小於操作符(<)的行為。
* \_\_gt\_\_(self, other)  
  定義大於操作符(>)的行為。
* \_\_le\_\_(self, other)  
  定義小於等於操作符(<)的行為。
* \_\_ge\_\_(self, other)  
  定義大於等於操作符(>)的行為。

舉個例子，假如我們想用一個類來存儲單詞。我們可能想按照字典序（字母順序）來比較單詞，字串的默認比較行為就是這樣。我們可能也想按照其他規則來比較字串，像是長度，或者音節的數量。在這個例子中，我們使用長度作為比較標準，下麵是一種實現:

**class** **Word**(str):

'''單詞類，按照單詞長度來定義比較行為'''

**def** **\_\_new\_\_**(cls, word):

*# 注意，我們只能使用 `\_\_new\_\_` ，因為str是不可變類型*

*# 所以我們必須提前初始化它（在實例創建時）*

**if** ' ' **in** word:

print "Value contains spaces. Truncating to first space."

word **=** word[:word**.**index(' ')]

*# Word現在包含第一個空格前的所有字母*

**return** str**.**`\_\_new\_\_`(cls, word)

**def** **\_\_gt\_\_**(self, other):

**return** len(self) **>** len(other)

**def** **\_\_lt\_\_**(self, other):

**return** len(self) **<** len(other)

**def** **\_\_ge\_\_**(self, other):

**return** len(self) **>=** len(other)

**def** **\_\_le\_\_**(self, other):

**return** len(self) **<=** len(other)

現在我們可以創建兩個 Word 對象（ Word('foo') 和 Word('bar'))然後根據長度來比較它們。注意我們沒有定義 \_\_eq\_\_ 和 \_\_ne\_\_ ，這是因為有時候它們會導致奇怪的結果（很明顯， Word('foo') == Word('bar') 得到的結果會是true）。根據長度測試是否相等毫無意義，所以我們使用 str 的實現來比較相等。

從上面可以看到，不需要實現所有的比較魔法方法，就可以使用豐富的比較操作。標準庫還在 functools 模組中提供了一個類裝飾器，只要我們定義 \_\_eq\_\_ 和另外一個操作符（ \_\_gt\_\_, \_\_lt\_\_ 等），它就可以幫我們實現比較方法。這個特性只在 Python 2.7 中可用。當它可用時，它能幫助我們節省大量的時間和精力。要使用它，只需要它 @total\_ordering 放在類的定義之上就可以了

### **2.2 數值操作符**

就像你可以使用比較操作符來比較類的實例，你也可以定義數值操作符的行為。固定好你的安全帶，這樣的操作符真的有很多。看在組織的份上，我把它們分成了五類：一元操作符，常見算數操作符，反射算數操作符（後面會涉及更多），增強賦值操作符，和類型轉換操作符。

### **一元操作符**

一元操作符只有一個操作符。

* \_\_pos\_\_(self)  
  實現取正操作，例如 +some\_object。
* \_\_neg\_\_(self)  
  實現取負操作，例如 -some\_object。
* \_\_abs\_\_(self)  
  實現內建絕對值函數 abs() 操作。
* \_\_invert\_\_(self)  
  實現取反操作符 ~。
* \_\_round\_\_(self， n)  
  實現內建函數 round() ，n 是近似小數點的位數。
* \_\_floor\_\_(self)  
  實現 math.floor() 函數，即向下取整。
* \_\_ceil\_\_(self)  
  實現 math.ceil() 函數，即向上取整。
* \_\_trunc\_\_(self)  
  實現 math.trunc() 函數，即距離零最近的整數。

### **常見算數操作符**

現在，我們來看看常見的二元操作符（和一些函數），像+，-，\*之類的，它們很容易從字面意思理解。

* \_\_add\_\_(self, other)  
  實現加法操作。
* \_\_sub\_\_(self, other)  
  實現減法操作。
* \_\_mul\_\_(self, other)  
  實現乘法操作。
* \_\_floordiv\_\_(self, other)  
  實現使用 // 操作符的整數除法。
* \_\_div\_\_(self, other)  
  實現使用 / 操作符的除法。
* \_\_truediv\_\_(self, other)  
  實現 *true* 除法，這個函數只有使用 from \_\_**future\_\_** import division 時才有作用。
* \_\_mod\_\_(self, other)  
  實現 % 取餘操作。
* \_\_divmod\_\_(self, other)  
  實現 divmod 內建函數。
* \_\_pow\_\_  
  實現 \*\* 操作符。
* \_\_lshift\_\_(self, other)  
  實現左移位運算符 << 。
* \_\_rshift\_\_(self, other)  
  實現右移位運算符 >> 。
* \_\_and\_\_(self, other)  
  實現按位與運算符 & 。
* \_\_or\_\_(self, other)  
  實現按位或運算符 | 。
* \_\_xor\_\_(self, other)  
  實現按位異或運算符 ^ 。

### **反射算數運算符**

還記得剛才我說會談到反射運算符嗎？可能你會覺得它是什麼高端霸氣上檔次的概念，其實這東西挺簡單的，下麵舉個例子:

some\_object **+** other

這是“常見”的加法，反射是一樣的意思，只不過是運算符交換了一下位置:

other **+** some\_object

所有反射運算符魔法方法和它們的常見版本做的工作相同，只不過是處理交換連個運算元之後的情況。絕大多數情況下，反射運算和正常順序產生的結果是相同的，所以很可能你定義 \_\_radd\_\_ 時只是調用一下 \_\_add\_\_。注意一點，操作符左側的對象（也就是上面的 other ）一定不要定義（或者產生 NotImplemented 異常） 操作符的非反射版本。例如，在上面的例子中，只有當 other 沒有定義 \_\_add\_\_ 時 some\_object.\_\_radd\_\_ 才會被調用。

* \_\_radd\_\_(self, other)  
  實現反射加法操作。
* \_\_rsub\_\_(self, other)  
  實現反射減法操作。
* \_\_rmul\_\_(self, other)  
  實現反射乘法操作。
* \_\_rfloordiv\_\_(self, other)  
  實現使用 // 操作符的整數反射除法。
* \_\_rdiv\_\_(self, other)  
  實現使用 / 操作符的反射除法。
* \_\_rtruediv\_\_(self, other)  
  實現 *true* 反射除法，這個函數只有使用 from \_\_future\_\_ import division時才有作用。
* \_\_rmod\_\_(self, other)  
  實現 % 反射取餘操作符。
* \_\_rdivmod\_\_(self, other)  
  實現調用 divmod(other, self) 時 divmod 內建函數的操作。
* \_\_rpow\_\_  
  實現 \*\* 反射操作符。
* \_\_rlshift\_\_(self, other)  
  實現反射左移位運算符 << 的作用。
* \_\_rshift\_\_(self, other)  
  實現反射右移位運算符 >> 的作用。
* \_\_rand\_\_(self, other)  
  實現反射按位與運算符 & 。
* \_\_ror\_\_(self, other)  
  實現反射按位或運算符 | 。
* \_\_rxor\_\_(self, other)  
  實現反射按位異或運算符 ^ 。

### **增強賦值運算符**

Python同樣提供了大量的魔法方法，可以用來自定義增強賦值操作的行為。或許你已經瞭解增強賦值，它融合了“常見”的操作符和賦值操作，如果你還是沒聽明白，看下麵的例子:

x **=** 5x **+=** 1 *# 也就是 x = x + 1*

這些方法都應該返回左側運算元應該被賦予的值（例如， a += b \_\_iadd\_\_ 也許會返回 a + b ，這個結果會被賦給 a ）,下麵是方法列表：

* \_\_iadd\_\_(self, other)  
  實現加法賦值操作。
* \_\_isub\_\_(self, other)  
  實現減法賦值操作。
* \_\_imul\_\_(self, other)  
  實現乘法賦值操作。
* \_\_ifloordiv\_\_(self, other)  
  實現使用 //= 操作符的整數除法賦值操作。
* \_\_idiv\_\_(self, other)  
  實現使用 /= 操作符的除法賦值操作。
* \_\_itruediv\_\_(self, other)  
  實現 *true* 除法賦值操作，這個函數只有使用 from \_\_**future\_\_** import division 時才有作用。
* \_\_imod\_\_(self, other)  
  實現 %= 取餘賦值操作。
* \_\_ipow\_\_  
  實現 \*\*= 操作。
* \_\_ilshift\_\_(self, other)  
  實現左移位賦值運算符 <<= 。
* \_\_irshift\_\_(self, other)  
  實現右移位賦值運算符 >>= 。
* \_\_iand\_\_(self, other)  
  實現按位與運算符 &= 。
* \_\_ior\_\_(self, other)  
  實現按位或賦值運算符 | 。
* **ixor**(self, other)  
  實現按位異或賦值運算符 ^= 。

### **類型轉換操作符**

Python也有一系列的魔法方法用於實現類似 float() 的內建類型轉換函數的操作。它們是這些：

* \_\_int\_\_(self)  
  實現到int的類型轉換。
* \_\_long\_\_(self)  
  實現到long的類型轉換。
* \_\_float\_\_(self)  
  實現到float的類型轉換。
* \_\_complex\_\_(self)  
  實現到complex的類型轉換。
* \_\_oct\_\_(self)  
  實現到八進制數的類型轉換。
* \_\_hex\_\_(self)  
  實現到十六進制數的類型轉換。
* \_\_index\_\_(self)  
  實現當對象用於切片運算式時到一個整數的類型轉換。如果你定義了一個可能會用於切片操作的數值類型，你應該定義 **index**。
* \_\_trunc\_\_(self)  
  當調用 math.trunc(self) 時調用該方法， **trunc** 應該返回 self 截取到一個整數類型（通常是long類型）的值。
* \_\_coerce\_\_(self)  
  該方法用於實現混合模式算數運算，如果不能進行類型轉換， **coerce** 應該返回 None 。反之，它應該返回一個二元組 self 和 other ，這兩者均已被轉換成相同的類型。

## **03. 類的表示**

使用字串來表示類是一個相當有用的特性。在Python中有一些內建方法可以返回類的表示，相對應的，也有一系列魔法方法可以用來自定義在使用這些內建函數時類的行為。

* \_\_str\_\_(self)  
  定義對類的實例調用 str() 時的行為。
* \_\_repr\_\_(self)  
  定義對類的實例調用 repr() 時的行為。str() 和 repr() 最主要的差別在於“目標用戶”。repr() 的作用是產生機器可讀的輸出（大部分情況下，其輸出可以作為有效的Python代碼），而 str() 則產生人類可讀的輸出。
* \_\_unicode\_\_(self)  
  定義對類的實例調用 unicode() 時的行為。unicode() 和 str() 很像，只是它返回unicode字串。注意，如果調用者試圖調用 str() 而你的類只實現了 \_\_unicode\_\_() ，那麼類將不能正常工作。所有你應該總是定義 \_\_str\_\_() ，以防有些人沒有閑情雅致來使用unicode。
* \_\_format\_\_(self)  
  定義當類的實例用於新式字串格式化時的行為，例如， "Hello, 0:abc!".format(a) 會導致調用 a.**format**("abc") 。當定義你自己的數值類型或字串類型時，你可能想提供某些特殊的格式化選項，這種情況下這個魔法方法會非常有用。
* \_\_hash\_\_(self)  
  定義對類的實例調用 hash() 時的行為。它必須返回一個整數，其結果會被用於字典中鍵的快速比較。同時注意一點，實現這個魔法方法通常也需要實現 \_\_eq\_\_ ，並且遵守如下的規則：a == b 意味著 hash(a) == hash(b)。
* \_\_nonzero\_\_(self)  
  定義對類的實例調用 bool() 時的行為，根據你自己對類的設計，針對不同的實例，這個魔法方法應該相應地返回True或False。
* \_\_dir\_\_(self)  
  定義對類的實例調用 dir() 時的行為，這個方法應該向調用者返回一個屬性列表。一般來說，沒必要自己實現 \_\_dir\_\_ 。但是如果你重定義了 \_\_getattr\_\_ 或者 \_\_getattribute\_\_ （下個部分會介紹），乃至使用動態生成的屬性，以實現類的互動式使用，那麼這個魔法方法是必不可少的。

到這裏，我們基本上已經結束了魔法方法指南中無聊並且例子匱乏的部分。既然我們已經介紹了較為基礎的魔法方法，是時候涉及更高級的內容了。

## **04. 訪問控制**

很多從其他語言轉向Python的人都抱怨Python的類缺少真正意義上的封裝（即沒辦法定義私有屬性然後使用公有的getter和setter）。然而事實並非如此。實際上Python不是通過顯式定義的字段和方法修改器，而是通過魔法方法實現了一系列的封裝。

* **getattr**(self, name)

當用戶試圖訪問一個根本不存在（或者暫時不存在）的屬性時，你可以通過這個魔法方法來定義類的行為。這個可以用於捕捉錯誤的拼寫並且給出指引，使用廢棄屬性時給出警告（如果你願意，仍然可以計算並且返回該屬性），以及靈活地處理AttributeError。只有當試圖訪問不存在的屬性時它才會被調用，所以這不能算是一個真正的封裝的辦法。

* **setattr**(self, name, value)

和 \_\_getattr\_\_ 不同， \_\_setattr\_\_ 可以用於真正意義上的封裝。它允許你自定義某個屬性的賦值行為，不管這個屬性存在與否，也就是說你可以對任意屬性的任何變化都定義自己的規則。然後，一定要小心使用 \_\_setattr\_\_ ，這個列表最後的例子中會有所展示。

* **delattr**(self, name)

這個魔法方法和 \_\_setattr\_\_幾乎相同，只不過它是用於處理刪除屬性時的行為。和 \_setattr\_\_ 一樣，使用它時也需要多加小心，防止產生無限遞歸（在 \_\_delattr\_\_的實現中調用 del self.name 會導致無限遞歸）。

* **getattribute**(self, name)

\_\_getattribute\_\_ 看起來和上面那些方法很合得來，但是最好不要使用它。\_\_getattribute\_\_ 只能用於新式類。在最新版的Python中所有的類都是新式類，在老版Python中你可以通過繼承 object 來創建新式類。\_\_getattribute\_\_ 允許你自定義屬性被訪問時的行為，它也同樣可能遇到無限遞歸問題（通過調用基類的 \_\_getattribute\_\_ 來避免）。\_\_getattribute\_\_ 基本上可以替代 \_\_getattr\_\_ 。只有當它被實現，並且顯式地被調用，或者產生 AttributeError 時它才被使用。這個魔法方法可以被使用（畢竟，選擇權在你自己），我不推薦你使用它，因為它的使用範圍相對有限（通常我們想要在賦值時進行特殊操作，而不是取值時），而且實現這個方法很容易出現Bug。

自定義這些控制屬性訪問的魔法方法很容易導致問題，考慮下麵這個例子:

**def** **\_\_setattr\_\_**(self, name**.** value):

self**.**name **=** value

*# 因為每次屬性幅值都要調用 \_\_setattr\_\_()，所以這裏的實現會導致遞歸*

*# 這裏的調用實際上是 self.\_\_setattr('name', value)。因為這個方法一直*

*# 在調用自己，因此遞歸將持續進行，直到程式崩潰*

**def** **\_\_setattr\_\_**(self, name, value):

self**.**\_\_dict\_\_[name] **=** value *# 使用 \_\_dict\_\_ 進行賦值*

*# 定義自定義行為*

再次重申，Python的魔法方法十分強大，能力越強責任越大，瞭解如何正確的使用魔法方法更加重要。

到這裏，我們對Python中自定義屬性存取控制有了什麼樣的印象？它並不適合輕度的使用。實際上，它有些過分強大，而且違反直覺。然而它之所以存在，是因為一個更大的原則：Python不指望讓杜絕壞事發生，而是想辦法讓做壞事變得困難。自由是至高無上的權利，你真的可以隨心所欲。下麵的例子展示了實際應用中某些特殊的屬性訪問方法（注意我們之所以使用 super 是因為不是所有的類都有 \_\_dict\_\_ 屬性）:

**class** **AccessCounter**(object):

''' 一個包含了一個值並且實現了訪問計數器的類 每次值的變化都會導致計數器自增'''

**def** **\_\_init\_\_**(self, val):

super(AccessCounter, self)**.**\_\_setattr\_\_('counter', 0)

super(AccessCounter, self)**.**\_\_setattr\_\_('value', val)

**def** **\_\_setattr\_\_**(self, name, value):

**if** name **==** 'value':

super(AccessCounter, self)**.**\_\_setattr\_('counter', self**.**counter **+** 1)

*# 使計數器自增變成不可避免*

*# 如果你想阻止其他屬性的賦值行為*

*# 產生 AttributeError(name) 就可以了*

super(AccessCounter, self)**.**\_\_setattr\_\_(name, value)

**def** **\_\_delattr\_\_**(self, name):

**if** name **==** 'value':

super(AccessCounter, self)**.**\_\_setattr('counter', self**.**counter **+** 1)

super(AccessCounter, self)**.**\_\_delattr(name)

## **05. 自定義序列**

有許多辦法可以讓你的Python類表現得像是內建序列類型（字典，元組，列表，字串等）。這些魔法方式是目前為止我最喜歡的。它們給了你難以置信的控制能力，可以讓你的類與一系列的全局函數完美結合。在瞭解激動人心的內容之前，首先你需要掌握一些預備知識。

既然講到創建自己的序列類型，就不得不說一說協議了。協議類似某些語言中的介面，裏面包含的是一些必須實現的方法。在Python中，協議完全是非正式的，也不需要顯式的聲明，事實上，它們更像是一種參考標準。

為什麼我們要講協議？因為在Python中實現自定義容器類型需要用到一些協議。首先，不可變容器類型有如下協議：想實現一個不可變容器，你需要定義 \_\_len\_\_ 和 \_\_getitem\_\_(後面會具體說明）。可變容器的協議除了上面提到的兩個方法之外，還需要定義 \_\_setitem\_\_ 和 \_\_delitem\_\_ 。最後，如果你想讓你的對象可以迭代，你需要定義 \_\_iter\_\_ ，這個方法返回一個迭代器。迭代器必須遵守迭代器協議，需要定義 \_\_iter\_\_ （返回它自己）和 next 方法。

### **5.1 容器背後的魔法方法**

* \_\_len\_\_(self)  
  返回容器的長度，可變和不可變類型都需要實現。
* \_\_getitem\_\_(self, key)  
  定義對容器中某一項使用 self[key] 的方式進行讀取操作時的行為。這也是可變和不可變容器類型都需要實現的一個方法。它應該在鍵的類型錯誤式產生 TypeError 異常，同時在沒有與鍵值相匹配的內容時產生 KeyError 異常。
* \_\_setitem\_\_(self, key)  
  定義對容器中某一項使用 self[key] 的方式進行賦值操作時的行為。它是可變容器類型必須實現的一個方法，同樣應該在合適的時候產生 KeyError 和 TypeError 異常。
* \_\_iter\_\_(self, key)  
  它應該返回當前容器的一個迭代器。迭代器以一連串內容的形式返回，最常見的是使用 iter() 函數調用，以及在類似 for x in container: 的迴圈中被調用。迭代器是他們自己的對象，需要定義 \_\_iter\_\_ 方法並在其中返回自己。
* \_\_reversed\_\_(self)  
  定義了對容器使用 reversed() 內建函數時的行為。它應該返回一個反轉之後的序列。當你的序列類是有序時，類似列表和元組，再實現這個方法，
* \_\_contains\_\_(self, item)  
  \_\_contains\_\_ 定義了使用 in 和 not in 進行成員測試時類的行為。你可能好奇為什麼這個方法不是序列協議的一部分，原因是，如果 **contains** 沒有定義，Python就會迭代整個序列，如果找到了需要的一項就返回 True 。
* \_\_missing\_\_(self ,key)  
  \_\_missing\_\_ 在字典的子類中使用，它定義了當試圖訪問一個字典中不存在的鍵時的行為（目前為止是指字典的實例，例如我有一個字典 d ， "george" 不是字典中的一個鍵，當試圖訪問 d["george'] 時就會調用 d.\_\_missing\_\_("george") ）。

### **5.2 一個例子**

讓我們來看一個實現了一些函數式結構的列表，可能在其他語言中這種結構更常見（例如Haskell）:

**class** **FunctionalList**:

'''一個列表的封裝類，實現了一些額外的函數式 方法，例如head, tail, init, last, drop和take。'''

**def** **\_\_init\_\_**(self, values**=None**):

**if** values **is** **None**:

self**.**values **=** []

**else**:

self**.**values **=** values

**def** **\_\_len\_\_**(self):

**return** len(self**.**values)

**def** **\_\_getitem\_\_**(self, key):

*# 如果鍵的類型或值不合法，列表會返回異常*

**return** self**.**values[key]

**def** **\_\_setitem\_\_**(self, key, value):

self**.**values[key] **=** value

**def** **\_\_delitem\_\_**(self, key):

**del** self**.**values[key]

**def** **\_\_iter\_\_**(self):

**return** iter(self**.**values)

**def** **\_\_reversed\_\_**(self):

**return** reversed(self**.**values)

**def** **append**(self, value):

self**.**values**.**append(value)

**def** **head**(self):

*# 取得第一個元素*

**return** self**.**values[0]

**def** **tail**(self):

*# 取得除第一個元素外的所有元素*

**return** self**.**valuse[1:]

**def** **init**(self):

*# 取得除最後一個元素外的所有元素*

**return** self**.**values[:**-**1]

**def** **last**(self):

*# 取得最後一個元素*

**return** self**.**values[**-**1]

**def** **drop**(self, n):

*# 取得除前n個元素外的所有元素*

**return** self**.**values[n:]

**def** **take**(self, n):

*# 取得前n個元素*

**return** self**.**values[:n]

就是這些，一個（微不足道的）有用的例子，向你展示了如何實現自己的序列。當然啦，自定義序列有更大的用處，而且絕大部分都在標準庫中實現了（Python是自帶電池的，記得嗎？），像 Counter , OrderedDict 和 NamedTuple 。

**06. 反射**

你可以通過定義魔法方法來控制用於反射的內建函數 isinstance 和 issubclass 的行為。下麵是對應的魔法方法：

* \_\_instancecheck\_\_(self, instance)  
  檢查一個實例是否是你定義的類的一個實例（例如 isinstance(instance, class) ）。
* \_\_subclasscheck\_\_(self, subclass)  
  檢查一個類是否是你定義的類的子類（例如 issubclass(subclass, class) ）。

這幾個魔法方法的適用範圍看起來有些窄，事實也正是如此。我不會在反射魔法方法上花費太多時間，因為相比其他魔法方法它們顯得不是很重要。但是它們展示了在Python中進行面向對象編程（或者總體上使用Python進行編程）時很重要的一點：不管做什麼事情，都會有一個簡單方法，不管它常用不常用。這些魔法方法可能看起來沒那麼有用，但是當你真正需要用到它們的時候，你會感到很幸運，因為它們還在那兒（也因為你閱讀了這本指南！）

## **07. 抽象基類**

請參考 *[http://docs.python.org/2/library/abc.html](https://link.zhihu.com/?target=http://docs.python.org/2/library/abc.html" \t "https://zhuanlan.zhihu.com/p/_blank)*

## **08. 可調用的對象**

你可能已經知道了，在Python中，函數是一等的對象。這意味著它們可以像其他任何對象一樣被傳遞到函數和方法中，這是一個十分強大的特性。

Python中一個特殊的魔法方法允許你自己類的對象表現得像是函數，然後你就可以“調用”它們，把它們傳遞到使用函數做參數的函數中，等等等等。這是另一個強大而且方便的特性，讓使用Python編程變得更加幸福。

* \_\_call\_\_ (self, [args…])  
  允許類的一個實例像函數那樣被調用。本質上這代表了 x() 和 x.\_\_call\_\_() 是相同的。注意 \_\_call\_\_ 可以有多個參數，這代表你可以像定義其他任何函數一樣，定義 \_\_call\_\_ ，喜歡用多少參數就用多少。

\_\_call\_\_ 在某些需要經常改變狀態的類的實例中顯得特別有用。“調用”這個實例來改變它的狀態，是一種更加符合直覺，也更加優雅的方法。一個表示平面上實體的類是一個不錯的例子:

**class** **Entity**:

'''表示一個實體的類，調用它的實例 可以更新實體的位置'''

**def** **\_\_init\_\_**(self, size, x, y):

self**.**x, self**.**y **=** x, y

self**.**size **=** size

**def** **\_\_call\_\_**(self, x, y):

'''改變實體的位置'''

self**.**x, self**.**y **=** x, y

## **09. 上下文管理器**

在Python 2.5中引入了一個全新的關鍵字，隨之而來的是一種新的代碼複用方法—— with 聲明。上下文管理的概念在Python中並不是全新引入的（之前它作為標準庫的一部分實現），直到PEP 343被接受，它才成為一種一級的語言結構。可能你已經見過這種寫法了:

**with** open('foo.txt') **as** bar:

*# 使用bar進行某些操作*

當對象使用 with 聲明創建時，上下文管理器允許類做一些設置和清理工作。上下文管理器的行為由下麵兩個魔法方法所定義：

* \_\_enter\_\_(self)  
  定義使用 with 聲明創建的語句塊最開始上下文管理器應該做些什麼。注意 \_\_enter\_\_ 的返回值會賦給 with 聲明的目標，也就是 as 之後的東西。
* \_\_exit\_\_(self, exception\_type, exception\_value, traceback)  
  定義當 with 聲明語句塊執行完畢（或終止）時上下文管理器的行為。它可以用來處理異常，進行清理，或者做其他應該在語句塊結束之後立刻執行的工作。如果語句塊順利執行， exception\_type , exception\_value 和 traceback 會是 None 。否則，你可以選擇處理這個異常或者讓用戶來處理。如果你想處理異常，確保 \_\_exit\_\_ 在完成工作之後返回 True 。如果你不想處理異常，那就讓它發生吧。

對一些具有良好定義的且通用的設置和清理行為的類，\_\_enter\_\_ 和 \_\_exit\_\_會顯得特別有用。你也可以使用這幾個方法來創建通用的上下文管理器，用來包裝其他對象。下麵是一個例子:

**class** **Closer**:

'''一個上下文管理器，可以在with語句中 使用close()自動關閉對象'''

**def** **\_\_init\_\_**(self, obj):

self**.**obj **=** obj

**def** **\_\_enter\_\_**(self, obj):

**return** self**.**obj *# 綁定到目標*

**def** **\_\_exit\_\_**(self, exception\_type, exception\_value, traceback):

**try**:

self**.**obj**.**close()

**except** **AttributeError**: *# obj不是可關閉的*

print 'Not closable.'

**return** **True** *# 成功地處理了異常*

這是一個 Closer 在實際使用中的例子，使用一個FTP連接來演示（一個可關閉的socket):

**>>>** **from** magicmethods **import** Closer**>>>** **from** ftplib **import** FTP**>>>** **with** Closer(FTP('ftp.somesite.com')) **as** conn:**...** conn**.**dir()**...***# 為了簡單，省略了某些輸出***>>>** conn**.**dir()*# 很長的 AttributeError 資訊，不能使用一個已關閉的連接***>>>** **with** Closer(int(5)) **as** i:**...** i **+=** 1**...**Not closable**.>>>** i6

看到我們的包裝器是如何同時優雅地處理正確和不正確的調用了嗎？這就是上下文管理器和魔法方法的力量。Python標準庫包含一個 contextlib 模組，裏面有一個上下文管理器 contextlib.closing() 基本上和我們的包裝器完成的是同樣的事情（但是沒有包含任何當對象沒有close()方法時的處理）。

## **10. 創建描述符對象**

描述符是一個類，當使用取值，賦值和刪除 時它可以改變其他對象。描述符不是用來單獨使用的，它們需要被一個擁有者類所包含。描述符可以用來創建面向對象資料庫，以及創建某些屬性之間互相依賴的類。描述符在表現具有不同單位的屬性，或者需要計算的屬性時顯得特別有用（例如表現一個坐標系中的點的類，其中的距離原點的距離這種屬性）。

要想成為一個描述符，一個類必須具有實現 \_\_get\_\_ , \_\_set\_\_ 和 \_\_delete\_\_ 三個方法中至少一個。

讓我們一起來看一看這些魔法方法：

* \_\_get\_\_(self, instance, owner)  
  定義當試圖取出描述符的值時的行為。instance 是擁有者類的實例， owner 是擁有者類本身。
* \_\_set\_\_(self, instance, owner)  
  定義當描述符的值改變時的行為。instance 是擁有者類的實例， value 是要賦給描述符的值。
* \_\_delete\_\_(self, instance, owner)  
  定義當描述符的值被刪除時的行為。instance 是擁有者類的實例

現在，來看一個描述符的有效應用：單位轉換:

**class** **Meter**(object):

'''米的描述符。'''

**def** **\_\_init\_\_**(self, value**=**0.0):

self**.**value **=** float(value)

**def** **\_\_get\_\_**(self, instance, owner):

**return** self**.**value

**def** **\_\_set\_\_**(self, instance, owner):

self**.**value **=** float(value)

**class** **Foot**(object):

'''英尺的描述符。'''

**def** **\_\_get\_\_**(self, instance, owner):

**return** instance**.**meter **\*** 3.2808

**def** **\_\_set\_\_**(self, instance, value):

instance**.**meter **=** float(value) **/** 3.2808

**class** **Distance**(object):

'''用於描述距離的類，包含英尺和米兩個描述符。'''

meter **=** Meter()

foot **=** Foot()

## **11. 拷貝**

有些時候，特別是處理可變對象時，你可能想拷貝一個對象，改變這個對象而不影響原有的對象。這時就需要用到Python的 copy 模組了。然而（幸運的是），Python模組並不具有感知能力， 因此我們不用擔心某天基於Linux的機器人崛起。但是我們的確需要告訴Python如何有效率地拷貝對象。

* \_\_copy\_\_(self)  
  定義對類的實例使用 copy.copy() 時的行為。copy.copy() 返回一個對象的淺拷貝，這意味著拷貝出的實例是全新的，然而裏面的數據全都是引用的。也就是說，對象本身是拷貝的，但是它的數據還是引用的（所以淺拷貝中的數據更改會影響原對象）。
* \_\_deepcopy\_\_(self, memodict=)  
  定義對類的實例使用 copy.deepcopy() 時的行為。copy.deepcopy() 返回一個對象的深拷貝，這個對象和它的數據全都被拷貝了一份。memodict 是一個先前拷貝對象的緩存，它優化了拷貝過程，而且可以防止拷貝遞歸數據結構時產生無限遞歸。當你想深拷貝一個單獨的屬性時，在那個屬性上調用 copy.deepcopy() ，使用 memodict 作為第一個參數。

這些魔法方法有什麼用武之地呢？像往常一樣，當你需要比默認行為更加精確的控制時。例如，如果你想拷貝一個對象，其中存儲了一個字典作為緩存（可能會很大），拷貝緩存可能是沒有意義的。如果這個緩存可以在內存中被不同實例共用，那麼它就應該被共用。

## **12. Pickling**

如果你和其他的Python愛好者共事過，很可能你已經聽說過Pickling了。Pickling是Python數據結構的序列化過程，當你想存儲一個對象稍後再取出讀取時，Pickling會顯得十分有用。然而它同樣也是擔憂和混淆的主要來源。

Pickling是如此的重要，以至於它不僅僅有自己的模組（ pickle ），還有自己的協議和魔法方法。首先，我們先來簡要的介紹一下如何pickle已存在的對象類型（如果你已經知道了，大可跳過這部分內容）。

### **12.1 小試牛刀**

我們一起來pickle吧。假設你有一個字典，你想存儲它，稍後再取出來。你可以把它的內容寫入一個檔，小心翼翼地確保使用了正確地格式，要把它讀取出來，你可以使用 exec() 或處理檔輸入。但是這種方法並不可靠：如果你使用純文本來存儲重要數據，數據很容易以多種方式被破壞或者修改，導致你的程式崩潰，更糟糕的情況下，還可能在你的電腦上運行惡意代碼。因此，我們要pickle它:

**import** pickle

data **=** {'foo': [1,2,3],

'bar': ('Hello', 'world!'),

'baz': **True**}jar **=** open('data.pkl', 'wb')pickle**.**dump(data, jar) *# 將pickle後的數據寫入jar檔*jar**.**close()

過了幾個小時，我們想把它取出來，我們只需要反pickle它:

**import** pickle

pkl\_file **=** open('data.pkl', 'rb') *# 與pickle後的數據連接*data **=** pickle**.**load(pkl\_file) *# 把它加載進一個變數*print datapkl\_file**.**close()

將會發生什麼？正如你期待的，它就是我們之前的 data 。

現在，還需要謹慎地說一句：pickle並不完美。Pickle檔很容易因為事故或被故意的破壞掉。Pickling或許比純文本檔安全一些，但是依然有可能被用來運行惡意代碼。而且它還不支持跨Python版本，所以不要指望分發pickle對象之後所有人都能正確地讀取。然而不管怎麼樣，它依然是一個強有力的工具，可以用於緩存和其他類型的持久化工作。

### **12.2 Pickle你的對象**

Pickle不僅僅可以用於內建類型，任何遵守pickle協議的類都可以被pickle。Pickle協議有四個可選方法，可以讓類自定義它們的行為（這和C語言擴展略有不同，那不在我們的討論範圍之內）。

* \_\_getinitargs\_\_(self)  
  如果你想讓你的類在反pickle時調用 \_\_init\_\_ ，你可以定義 \_\_getinitargs\_\_(self) ，它會返回一個參數元組，這個元組會傳遞給 \_\_init\_\_。注意，這個方法只能用於舊式類。
* \_\_getnewargs\_\_(self)  
  對新式類來說，你可以通過這個方法改變類在反pickle時傳遞給 \_\_new\_\_ 的參數。這個方法應該返回一個參數元組。
* \_\_getstate\_\_(self)  
  你可以自定義對象被pickle時被存儲的狀態，而不使用對象的 \_\_dict\_\_ 屬性。這個狀態在對象被反pickle時會被 \_\_setstate\_\_ 使用。
* \_\_setstate\_\_(self)  
  當一個對象被反pickle時，如果定義了 \_\_setstate\_\_ ，對象的狀態會傳遞給這個魔法方法，而不是直接應用到對象的 \_\_dict\_\_ 屬性。這個魔法方法和 \_\_getstate\_\_ 相互依存：當這兩個方法都被定義時，你可以在Pickle時使用任何方法保存對象的任何狀態。
* \_\_reduce\_\_(self)  
  當定義擴展類型時（也就是使用Python的C語言API實現的類型），如果你想pickle它們，你必須告訴Python如何pickle它們。**reduce** 被定義之後，當對象被Pickle時就會被調用。它要麼返回一個代表全局名稱的字串，Pyhton會查找它並pickle，要麼返回一個元組。這個元組包含2到5個元素，其中包括：一個可調用的對象，用於重建對象時調用；一個參數元素，供那個可調用對象使用；被傳遞給 \_\_setstate\_\_ 的狀態（可選）；一個產生被pickle的列表元素的迭代器（可選）；一個產生被pickle的字典元素的迭代器（可選）；
* \_\_reduce\_ex\_\_(self)  
  \_\_reduce\_ex\_\_ 的存在是為了相容性。如果它被定義，在pickle時 \_\_reduce\_ex\_\_ 會代替 \_\_reduce\_\_ 被調用。\_\_reduce\_\_ 也可以被定義，用於不支持 \_\_reduce\_ex\_\_ 的舊版pickle的API調用。

### **12.3 一個例子**

我們的例子是 Slate ，它會記住它的值曾經是什麼，以及那些值是什麼時候賦給它的。然而 每次被pickle時它都會變成空白，因為當前的值不會被存儲:

**import** time

**class** **Slate**:

'''存儲一個字串和一個變更日誌的類 每次被pickle都會忘記它當前的值'''

**def** **\_\_init\_\_**(self, value):

self**.**value **=** value

self**.**last\_change **=** time**.**asctime()

self**.**history **=** {}

**def** **change**(self, new\_value):

*# 改變當前值，將上一個值記錄到歷史*

self**.**history[self**.**last\_change] **=** self**.**value

self**.**value **=** new\_value)

self**.**last\_change **=** time**.**asctime()

**def** **print\_change**(self):

print 'Changelog for Slate object:'

**for** k,v **in** self**.**history**.**items():

print '%s\t %s' **%** (k,v)

**def** **\_\_getstate\_\_**(self):

*# 故意不返回self.value或self.last\_change*

*# 我們想在反pickle時得到一個空白的slate*

**return** self**.**history

**def** **\_\_setstate\_\_**(self):

*# 使self.history = slate，last\_change*

*# 和value為未定義*

self**.**history **=** state

self**.**value, self**.**last\_change **=** **None**, **None**

## **魔法方法**

如果你已經使用了一段時間Python，那麼一定瞭解或者接觸過魔法方法。

魔法方法是一種以**雙下劃線**開頭和結尾的一種特殊方法，在使用類的時候非常常見，例如，經常會用到的\_\_init\_\_。它的功能是作為構造函數，能夠在類初始化時調用，你可以在初始化方法中定義一些初始化變數、初始化操作，當執行到類內部時，它會首先執行這些方法。

當然，魔法方法遠不至於\_\_init\_\_，為了幫助大家理解Python魔法方法的價值，本文就以一個示例來開始本文的講解。

下麵通過實現一個名為TimePeriod的類，來看一下如何使用Python魔法方法使得代碼更加清晰、可讀性更高。

## **基礎的TimePeriod類**

下麵TimePeriod主要實現2個方法：

* 時間的增加
* 時間的比較

基礎的實現方法大多會是下麵這樣：

**class** **TimePeriod**:

**def** \_\_init\_\_(self, hours**=**0, minutes**=**0):

self**.**hours **=** hours

self**.**minutes **=** minutes

**def** **add**(self, other):

minutes **=** self**.**minutes **+** other**.**minutes

hours **=** self**.**hours **+** other**.**hours

**if** minutes **>=** 60:

minutes **-=** 60

hours **+=** 1

**return** TimePeriod(hours, minutes)

**def** **greater\_than**(self, other):

**if** self**.**hours **>** other**.**hours:

returnTrue

**elif** self**.**hours **<** other**.**hours:

returnFalse

**elif** self**.**minutes **>** other**.**minutes:

returnTrue

**else**:

returnFalse

實現add和greater\_than兩個方法，分別用於增加和對比時間大小。

或許，從這段代碼中看不出有任何問題。接下來，我們使用這個類來看一下。

time\_i\_sleep = TimePeriod(9, 0)

time\_i\_work = TimePeriod(0, 30)

print(time\_i\_sleep.greater\_than(time\_i\_work))

# True

這段代碼在執行的時候沒有任何問題，也不會報錯。但是，如果你想要執行更為複雜的操作，例如，2個時間段**加和**再和另外2個時間段的**加和**進行對比，類似於A+B與C+D進行比較，如果使用上述這個類，對比會是這樣的：

time\_i\_sleep.add(time\_i\_watch\_netflix).greater\_than(time\_i\_work.add(time\_i\_do\_chores))

這樣看上去是不是非常複雜？

即便是作為一名聰明的開發人員，把這段代碼拆開執行，也會像下麵這樣複雜：

time\_spent\_unproductively **=** time\_i\_sleep**.**add(time\_i\_watch\_netflix)time\_spent\_productively **=** time\_i\_work**.**add(time\_i\_do\_chores)time\_spent\_unproductively**.**greater\_than(time\_spent\_productively)

## **魔法方法實現方式**

其實在Python中，很容易就可以實現上述功能，Python內置的有2個魔方方法\_\_add\_\_和\_\_gt\_\_分別對應於+和>運算。

現在，通過魔法方法來修改一下上面的類：

**class** **TimePeriod**:

**def** \_\_init\_\_(self, hours**=**0, minutes**=**0):

self**.**hours **=** hours

self**.**minutes **=** minutes

**def** \_\_add\_\_(self, other):

minutes **=** self**.**minutes **+** other**.**minutes

hours **=** self**.**hours **+** other**.**hours

**if** minutes **>=** 60:

minutes **-=** 60

hours **+=** 1

**return** TimePeriod(hours, minutes)

**def** \_\_gt\_\_(self, other):

**if** self**.**hours **>** other**.**hours:

returnTrue

**elif** self**.**hours **<** other**.**hours:

returnFalse

**elif** self**.**minutes **>** other**.**minutes:

returnTrue

**else**:

returnFalse

現在，再來進行一下對比運算：

(time\_i\_sleep + time\_i\_watch\_netflix) > (time\_i\_work + time\_i\_do\_chores)

這樣看起來是不是更加容易理解和閱讀了？

除此之外，你還可以加入更多魔法方法，實現更為豐富的功能。

例如，你想比較2個時間是否相等，會用到==運算，這時候你可以使用魔法方法\_\_eq\_\_，具體實現如下：

**def** \_\_eq\_\_(self, other):

**return** self**.**hours **==** other**.**hours **and** self**.**minutes **==** other**.**minutes

Python魔法方法不止有算術運算和和比較運算，還有其他很多豐富的功能。例如，\_\_str\_\_，可以創建易於理解類的字串。

**def** \_\_str\_\_(self):

returnf"{self.hours} hours, {self.minutes} minutes"

如果你需要，還可以通過\_\_getitem\_\_把類轉化成字典：

**def** \_\_getitem\_\_(self, item):

**if** item **==** 'hours':

**return** self**.**hours

**elif** item **==** 'minutes':

**return** self**.**minutes

**else**:

**raise** **KeyError**()

這樣，可以把很多字典的優質特性加入到類中，可以像訪問字典一樣去訪問類的屬性。

## **其他**

除了上述提到的一些魔法方法，Python還有很多魔法方法值得使用。本文不再逐一舉例，下麵介紹一下一些常用方法的功能，需要的可以在以後編碼中用一下。

* \_\_new\_\_：初始化類的實例時會調用\_\_init\_\_方法，而在實際創建實例時會更早地調用\_\_new\_\_方法。
* \_\_call\_\_：\_\_call\_\_方法允許我們的實例像方法或函數一樣可調用。
* \_\_len\_\_：這允許你自定義Python內置len()函數。
* \_\_repr\_\_：這類似於\_\_str\_\_ 魔法方法，它允許你定義類的字串表示形式。但是，區別在於\_\_str\_\_是針對最終用戶的，它提供了一個更加用戶友好的非正式字串，而\_\_repr\_\_是針對開發人員的，並且可能包含有關類的內部狀態的更複雜的資訊。
* \_\_setitem\_\_：前面示例中我們已經看過\_\_getitem\_\_方法，它主要用於獲取鍵值，而\_\_setitem\_\_則用於設置鍵值。
* \_\_enter\_\_和\_\_exit\_\_：這兩種方法通常一起使用，可以將你的類用作上下文管理器，實現類似Python中with語句的功能。