**PYTHON 學習筆記**

list 轉 tuple 範例 2

tuple 轉 list範例 2

list 轉 dict 範例 2

dict 轉 list 範例 3

tuple 轉 dict範例 3

dict 轉 tuple範例 4

list 的高级用法 5

dict高级用法 7

for 的高级用法 10

while 的高级用法 12

模擬 goto 用法 13

set 的高级用法 15

模擬 switch/case 用法 16

函式 的高级用法 18

Python 內建物件 21

\_\_init\_\_ ()與 \_\_new\_\_ ()區別 24

變量查找機制 25

MRO（方法解析順序）與多繼承 26

**描述符機制** 27

\_\_getattr\_\_()\_\_getattribute\_\_()

property 裝飾器 29

元類（metaclass） 30

物件協定（Object Protocol） 31

操作符重載 32

迭代器協議 33

生成器(Generator) 34

yield 使用說明 38

**協程、生成器、Greenlet、執行緒和進程**

全域解譯器鎖 44

淺拷貝和深拷貝 45

ConfigParse 模組 46

argparse 模組 47

ElementTree 模組 49

JSON 模組 51

traceback 模組 52

logging 模組 53

threading 模組 55

Queue 模組 59

range 的高级用法 62

eval 的高級用法 64

format() 使用方法及示例 66

f-string格式化字串 71

模組的單例模式 74

使用 Mixin 模式 76

== 和 is 的適用場景 78

推導式指南 79

datetime 模組 81

可變參數 的高級用法 83

匿名函數 86

裝飾器(Decorator) 89

閉包(Closure) 92

魔法函數 94

\_\_init\_\_.py 的高級用法 121

**list 轉 tuple 請範例**

1. **將整個列表轉換為元組**

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

my\_tuple = tuple(my\_list)

print(my\_tuple) *# 輸出: (1, 2, 3, 4, 5)*

1. **將列表中的每個元素轉換為一個單獨的元組**

my\_list = ['apple', 'banana', 'cherry']

my\_tuples = [tuple(item) for item in my\_list]

print(my\_tuples) *# 輸出: [('apple',), ('banana',), ('cherry',)]*

1. **使用\*運算符將列表轉換為元組**

my\_list = [10, 20, 30]

my\_tuple = (10, 20, 30)

print(my\_tuple) *# 輸出: (10, 20, 30)*

my\_tuple = \*my\_list, *# 使用\*運算符將列表轉換為元組*

print(my\_tuple) *# 輸出: (10, 20, 30)*

1. **使用zip()函數將兩個列表轉換為一個元組列表**

names = ['Alice', 'Bob', 'Charlie']

ages = [25, 30, 35]

people = list(zip(names, ages))

print(people) *# 輸出: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]*

1. **使用map()函數將列表中的每個元素都轉換為元組**

my\_tuples = list(map(tuple, my\_list))

print(my\_tuples) *# 輸出: [('a',), ('b',), ('c',), ('d',)]*

**tuple 轉 list範例**

1. **將整個元組轉換為列表**

my\_tuple = (1, 2, 3, 4, 5)

my\_list = list(my\_tuple)

print(my\_list) *# 輸出: [1, 2, 3, 4, 5]*

1. **將元組中的每個元素轉換為單獨的列表**

my\_tuple = ('apple', 'banana', 'cherry')

my\_lists = [list(item) for item in my\_tuple]

print(my\_lists) *# 輸出: [['a', 'p', 'p', 'l', 'e'], ['b', 'a', 'n', 'a', 'n', 'a'], ['c', 'h', 'e', 'r', 'r', 'y']]*

1. **使用\*運算符將元組轉換為列表**

my\_tuple = (10, 20, 30)

my\_list = [10, 20, 30]

print(my\_list) *# 輸出: [10, 20, 30]*

my\_list = list(\*my\_tuple) *# 使用\*運算符將元組轉換為列表*

print(my\_list) *# 輸出: [10, 20, 30]*

1. **使用zip()函數將兩個元組轉換為一個列表**

names = ('Alice', 'Bob', 'Charlie')

ages = (25, 30, 35)

people = list(zip(names, ages))

print(people) *# 輸出: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]*

1. **使用map()函數將元組中的每個元素都轉換為列表**

my\_tuple = ('a', 'b', 'c', 'd')

my\_lists = list(map(list, my\_tuple))

print(my\_lists) *# 輸出: [['a'], ['b'], ['c'], ['d']]*

**list 轉 dict 範例**

1. **使用字典理解式(dict comprehension)**

*# 列表中的每個元素都是一對鍵值對*

pairs = [('apple', 1), ('banana', 2), ('orange', 3)]

my\_dict = {k: v for k, v in pairs}

print(my\_dict) *# 輸出: {'apple': 1, 'banana': 2, 'orange': 3}*

1. **使用zip()函數和兩個列表**

keys = ['a', 'b', 'c']

values = [1, 2, 3]

my\_dict = dict(zip(keys, values))

print(my\_dict) *# 輸出: {'a': 1, 'b': 2, 'c': 3}*

1. **使用dict()構造函數和列表中的元組**

pairs = [('name', 'Alice'), ('age', 25), ('city', 'New York')]

my\_dict = dict(pairs)

print(my\_dict) *# 輸出: {'name': 'Alice', 'age': 25, 'city': 'New York'}*

1. **使用enumerate()函數和列表**

colors = ['red', 'green', 'blue']

my\_dict = dict(enumerate(colors))

print(my\_dict) *# 輸出: {0: 'red', 1: 'green', 2: 'blue'}*

1. **使用映射函數(mapping function)**

people = [{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}]

my\_dict = dict((p['name'], p['age']) for p in people)

print(my\_dict) *# 輸出: {'Alice': 25, 'Bob': 30}*

1. **使用fromkeys()方法**

keys = ['a', 'b', 'c']

my\_dict = dict.fromkeys(keys, 0) *# 所有鍵對應的值都是0*

print(my\_dict) *# 輸出: {'a': 0, 'b': 0, 'c': 0}*

**dict 轉 list 範例**

1. **將字典的鍵轉換為列表**

my\_dict = {'apple': 1, 'banana': 2, 'orange': 3}

keys\_list = list(my\_dict.keys())

print(keys\_list) *# 輸出: ['apple', 'banana', 'orange']*

1. **將字典的值轉換為列表**

my\_dict = {'apple': 1, 'banana': 2, 'orange': 3}

values\_list = list(my\_dict.values())

print(values\_list) *# 輸出: [1, 2, 3]*

1. **將字典的鍵值對轉換為列表的元組**

my\_dict = {'apple': 1, 'banana': 2, 'orange': 3}

items\_list = list(my\_dict.items())

print(items\_list) *# 輸出: [('apple', 1), ('banana', 2), ('orange', 3)]*

1. **使用列表理解式(list comprehension)**

my\_dict = {'Alice': 25, 'Bob': 30, 'Charlie': 35}

names\_list = [name for name in my\_dict.keys()]

ages\_list = [age for age in my\_dict.values()]

items\_list = [(name, age) for name, age in my\_dict.items()]

print(names\_list) *# 輸出: ['Alice', 'Bob', 'Charlie']*

print(ages\_list) *# 輸出: [25, 30, 35]*

print(items\_list) *# 輸出: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]*

1. **使用zip()函數**

my\_dict = {'a': 1, 'b': 2, 'c': 3}

keys\_list = list(my\_dict.keys())

values\_list = list(my\_dict.values())

items\_list = list(zip(keys\_list, values\_list))

print(keys\_list) *# 輸出: ['a', 'b', 'c']*

print(values\_list) *# 輸出: [1, 2, 3]*

print(items\_list) *# 輸出: [('a', 1), ('b', 2), ('c', 3)]*

**tuple 轉 dict範例**

1. **使用dict()構造函數**

tuples = [('name', 'Alice'), ('age', 25), ('city', 'New York')]

my\_dict = dict(tuples)

print(my\_dict) *# 輸出: {'name': 'Alice', 'age': 25, 'city': 'New York'}*

1. **使用zip()函數和兩個元組**

keys = ('a', 'b', 'c')

values = (1, 2, 3)

my\_dict = dict(zip(keys, values))

print(my\_dict) *# 輸出: {'a': 1, 'b': 2, 'c': 3}*

1. **使用字典理解式(dict comprehension)**

my\_dict = {k: v for k, v in tuples}

print(my\_dict) *# 輸出: {'apple': 1, 'banana': 2, 'orange': 3}*

1. **使用enumerate()函數和單個元組**

colors = ('red', 'green', 'blue')

my\_dict = dict(enumerate(colors))

print(my\_dict) *# 輸出: {0: 'red', 1: 'green', 2: 'blue'}*

1. **使用zip()函數和多個元組**

keys = ('name', 'age', 'city')

values1 = ('Alice', 25)

values2 = ('New York',)

my\_dict = dict(zip(keys, values1 + values2))

print(my\_dict) *# 輸出: {'name': 'Alice', 'age': 25, 'city': 'New York'}*

在上述範例中,我們利用了dict()構造函數、zip()函數、字典理解式和enumerate()函數等方法將元組轉換為字典。

**dict 轉 tuple範例**

1. **將字典的鍵轉換為元組**

my\_dict = {'apple': 1, 'banana': 2, 'orange': 3}

keys\_tuple = tuple(my\_dict.keys())

print(keys\_tuple) *# 輸出: ('apple', 'banana', 'orange')*

1. **將字典的值轉換為元組**

my\_dict = {'apple': 1, 'banana': 2, 'orange': 3}

values\_tuple = tuple(my\_dict.values())

print(values\_tuple) *# 輸出: (1, 2, 3)*

1. **將字典的鍵值對轉換為元組**

my\_dict = {'apple': 1, 'banana': 2, 'orange': 3}

items\_tuple = tuple(my\_dict.items())

print(items\_tuple) *# 輸出: (('apple', 1), ('banana', 2), ('orange', 3))*

1. **使用列表推導式(list comprehension)和tuple()**

my\_dict = {'Alice': 25, 'Bob': 30, 'Charlie': 35}

names\_tuple = tuple(name for name in my\_dict.keys())

ages\_tuple = tuple(age for age in my\_dict.values())

items\_tuple = tuple((name, age) for name, age in my\_dict.items())

print(names\_tuple) *# 輸出: ('Alice', 'Bob', 'Charlie')*

print(ages\_tuple) *# 輸出: (25, 30, 35)*

print(items\_tuple) *# 輸出: (('Alice', 25), ('Bob', 30), ('Charlie', 35))*

1. **使用zip()函數和tuple()**

my\_dict = {'a': 1, 'b': 2, 'c': 3}

keys\_list = list(my\_dict.keys())

values\_list = list(my\_dict.values())

keys\_tuple = tuple(keys\_list)

values\_tuple = tuple(values\_list)

items\_tuple = tuple(zip(keys\_tuple, values\_tuple))

print(keys\_tuple) *# 輸出: ('a', 'b', 'c')*

print(values\_tuple) *# 輸出: (1, 2, 3)*

print(items\_tuple) *# 輸出: (('a', 1), ('b', 2), ('c', 3))*

**list 的高级用法**

1. **列表推導式(List Comprehension)**

這是一種簡潔的方式來創建列表,可以替代傳統的 for 循環。

squares = [x\*\*2 for x in range(10)] *# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]*

even\_squares = [x\*\*2 for x in range(10) if x % 2 == 0] *# [0, 4, 16, 36, 64]*

1. **嵌套列表推導式**

可以使用多個 for 循環創建嵌套列表。

matrix = [[j for j in range(5)] for i in range(3)]

*# [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]*

1. **列表解析(List Unpacking)**

可以將可迭代對象拆分為單獨的元素,賦值給列表。

a, \*b, c = [1, 2, 3, 4, 5]

print(a) *# 1*

print(b) *# [2, 3, 4]*

print(c) *# 5*

1. **切片(Slicing)**

可以使用切片語法來獲取列表的子序列。

my\_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(my\_list[2:8]) *# [2, 3, 4, 5, 6, 7]*

print(my\_list[::-1]) *# [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]*

1. **列表方法**

列表提供了許多實用的方法,如 append()、insert()、remove()、sort()、reverse() 等。

my\_list = [1, 3, 5]

my\_list.append(7) *# [1, 3, 5, 7]*

my\_list.insert(1, 4) *# [1, 4, 3, 5, 7]*

my\_list.remove(3) *# [1, 4, 5, 7]*

my\_list.sort(reverse=True) *# [7, 5, 4, 1]*

1. **列表作為堆疊(Stack)使用**

列表可以充當堆疊(LIFO)數據結構,使用 append() 和 pop() 方法。

stack = []

stack.append(1)

stack.append(2)

stack.append(3)

print(stack.pop()) *# 3*

print(stack.pop()) *# 2*

print(stack.pop()) *# 1*

1. **列表作為隊列(Queue)使用**

使用 append() 和 pop(0) 方法,列表也可以充當隊列(FIFO)數據結構。

queue = []

queue.append(1)

queue.append(2)

queue.append(3)

print(queue.pop(0)) *# 1*

print(queue.pop(0)) *# 2*

print(queue.pop(0)) *# 3*

**multi-list 的高級用法**

在 Python 中，多維列表（multi-list）是指包含多個列表的數據結構。它們可以用於存儲複雜的數據，例如多個維度的數據或具有不同數據類型的數據。

以下是 Python 中多列表的高級用法：

* **列表生成**：用於創建樹形結構或表示具有多個級別的數據

buf= [[0 for x in range(2)] for y in range(5)] # [[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]

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

buf= [[(x) for x in range(2)] for y in range(5)] # [[0, 1], [0, 1], [0, 1], [0, 1], [0, 1]]

buf= [[(y) for x in range(2)] for y in range(5)] # [[0, 0], [1, 1], [2, 2], [3, 3], [4, 4]]

buf= [[(x,y) for x in range(2)] for y in range(5)] # [[(0, 0), (1, 0)], [(0, 1), (1, 1)], [(0, 2), (1, 2)], [(0, 3), (1, 3)], [(0, 4), (1, 4)]]

buf= [[y\*(x==0) for x in range(2)] for y in range(5)] # [[0, 0], [1, 0], [2, 0], [3, 0], [4, 0]]

buf= [[[z for x in range(2)] for y in range(3)] for z in range(4)] #

[[[0, 0], [0, 0], [0, 0]],

[[1, 1], [1, 1], [1, 1]],

[[2, 2], [2, 2], [2, 2]],

[[3, 3], [3, 3], [3, 3]]]

* **巢狀列表**：嵌套列表是指包含其他列表的列表。這可以用於創建樹形結構或表示具有多個級別的數據。

data = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9],

]

for row in data:

for element in row:

print(element)

* **列表推導式**：列表推導式是一種簡潔的方法來創建列表。它們可以用於從其他列表生成新列表。

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

squares = [x \* x for row in data for x in row]

print(squares) # 輸出：[1, 4, 9, 16, 25, 36, 49, 64, 81]

* **列表解析**：列表解析是一種更通用的方法來創建列表。它們可以用於從其他列表或反覆運算器生成新列表。

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

even\_squares = [x \* x for row in data for x in row if x % 2 == 0]

print(even\_squares) # 輸出：[4, 16, 36, 64]

* **zip() 函式**：zip() 函式可用於將多個清單中的元素組合在一起。這可以用於創建並行列表或將數據從多個列表組合在一起。

names = ["Alice", "Bob", "Charlie"]

ages = [30, 25, 22]

profiles = zip(names, ages)

for name, age in profiles:

print(f"{name} is {age} years old.")

* **map() 函式**：map() 函式可用於將函式應用於清單中的每個元素。這可以用於對清單中的所有元素執行相同的操作。

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

doubles = map(lambda x: x \* 2, data)

print(list(doubles)) # 輸出：[2, 4, 6, 8, 10]

* **filter() 函式**：filter() 函式可用於從清單中篩選元素。這可以用於僅保留符合特定條件的元素。

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

evens = filter(lambda x: x % 2 == 0, data)

print(list(evens)) # 輸出：[2, 4]

* **reduce() 函式**：reduce() 函式可用於將清單中的元素組合在一起。這可以用於計算總和、平均值或其他聚合統計量。

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

total = reduce(lambda x, y: x + y, data)

print(total) # 輸出：15

**python 中 dict高级用法**

Python 中的字典(dict)是一種非常強大和靈活的數據結構,除了基本的鍵值對存儲和訪問功能外,還有一些高級用法值得了解:

1. **字典推導式(Dictionary Comprehension)**

這類似於列表和集合推導式,可以快速創建字典。

squares = {x: x\*\*2 for x in range(6)} *# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}*

1. **合併字典**

Python 3.9後可以使用 | 運算符合併字典,在此之前可以使用 update() 方法。

dict1 = {'a': 1, 'b': 2}

dict2 = {'c': 3, 'd': 4}

merged = dict1 | dict2 *# Python 3.9+: {'a': 1, 'b': 2, 'c': 3, 'd': 4}*

dict1.update(dict2) *# 將 dict2 合併到 dict1 中*

1. **defaultdict**

defaultdict 是 dict 的一個子類,可以為缺失的鍵提供一個默認值。

from collections import defaultdict

fruits = defaultdict(int) *# 缺失的鍵將初始化為0*

fruits['apple'] += 1

fruits['banana'] += 2

print(fruits) *# defaultdict(<class 'int'>, {'apple': 1, 'banana': 2})*

1. **計數器(Counter)**

Counter 是一個子類,用於計算可哈希對象的出現次數。

from collections import Counter

colors = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])

print(colors) *# Counter({'blue': 3, 'red': 2, 'green': 1})*

1. **有序字典(OrderedDict)**

OrderedDict 是 dict 的子類,記錄了鍵值對插入的順序。

from collections import OrderedDict

od = OrderedDict()

od['a'] = 1

od['b'] = 2

od['c'] = 3

print(od) *# OrderedDict([('a', 1), ('b', 2), ('c', 3)])*

1. **視圖對象(View Objects)**

視圖對象提供了一種高效且動態的方式來獲取字典的鍵、值或鍵值對。

my\_dict = {'a': 1, 'b': 2, 'c': 3}

keys\_view = my\_dict.keys() *# dict\_keys(['a', 'b', 'c'])*

values\_view = my\_dict.values() *# dict\_values([1, 2, 3])*

items\_view = my\_dict.items() *# dict\_items([('a', 1), ('b', 2), ('c', 3)])*

1. **散列和集合操作**

字典支持許多集合操作,如聯合、交集、差集等。

dict1 = {'a': 1, 'b': 2}

dict2 = {'b': 3, 'c': 4}

common = dict1.keys() & dict2.keys() *# {'b'}*

all\_keys = dict1.keys() | dict2.keys() *# {'a', 'c', 'b'}*

**8.將函數作為 dict 的 value,實現鍵值對應的功能映射**

*# 定義一些函數*

def square(x):

return x \*\* 2

def cube(x):

return x \*\* 3

*# 將函數作為 value 存入 dict*

operations = {

'square': square,

'cube': cube

}

*# 使用時直接呼叫對應的函數*

print(operations['square'](3)) *# 輸出: 9*

print(operations['cube'](3)) *# 輸出: 27*

9.**配合 lambda 使用,實現簡單的回調功能**

*# 定義一個回調函數*

callbacks = {

'square': lambda x: x \*\* 2,

'cube': lambda x: x \*\* 3

}

*# 定義高階函數,接受回調函數作為參數*

def apply\_operation(operation, value, callback):

result = operation(value)

return callback(result)

print(apply\_operation(lambda x: x \* 2, 3, callbacks['square'])) *# 輸出: 36*

print(apply\_operation(lambda x: x \* 2, 3, callbacks['cube'])) *# 輸出: 216*

10.**實現簡單的策略模式**

*# 定義一些策略函數*

def strategy\_a(data):

*# 執行策略 A 的邏輯...*

return f'Strategy A: {data}'

def strategy\_b(data):

*# 執行策略 B 的邏輯...*

return f'Strategy B: {data}'

*# 將策略函數存入 dict*

strategies = {

'a': strategy\_a,

'b': strategy\_b

}

*# 根據鍵選擇策略*

strategy\_key = 'a'

strategy = strategies[strategy\_key]

result = strategy('some data')

print(result) *# 輸出: Strategy A: some data*

11.**實現簡單的狀態機**

*# 定義狀態轉換函數*

def state\_a(data):

print(f'Current state: A, Data: {data}')

return 'b'

def state\_b(data):

print(f'Current state: B, Data: {data}')

return 'a'

*# 將狀態轉換函數存入 dict*

state\_machine = {

'a': state\_a,

'b': state\_b

}

*# 執行狀態機*

current\_state = 'a'

data = 'hello'

while True:

current\_state = state\_machine[current\_state](data)

**12. 使用函數作為默認值**

dict 的默認值可以用函數來代替。這意味著，如果鍵不存在於 dict 中，則將調用函數來生成值。例如，以下代碼創建一個 dict，其中默認值是一個生成隨機數的函數：

import random

def random\_value():

return random.randint(1, 100)

defaults = {

'key1': random\_value,

'key2': 50,

}

print(defaults['key1']) # 可能輸出：37

print(defaults['key2']) # 輸出：50

**13. 使用函數根據鍵生成值**

dict 的值可以用函數來生成，具體取決於鍵。這意味著，對於每個鍵，函數將被調用一次，並使用鍵作為參數來生成值。例如，以下代碼創建一個 dict，其中值是鍵的平方：

def square(x):

return x \* x

squares = {key: square(key) for key in range(10)}

print(squares[3]) # 輸出：9

print(squares[5]) # 輸出：25

**14. 使用函數惰性計算值**

dict 的值可以用函數來惰性計算。這意味著，只有當值實際需要時才會調用函數。這對於計算昂貴或耗時的值的函數很有用。例如，以下代碼創建一個 dict，其中值是計算文件行數的函數：

def count\_lines(filename):

with open(filename) as f:

return sum(1 for line in f)

file\_counts = {filename: count\_lines for filename in ['data1.txt', 'data2.txt']}

print(file\_counts['data1.txt']) # 實際調用 count\_lines() 時才輸出行數

print(file\_counts['data2.txt']) # 實際調用 count\_lines() 時才輸出行數

使用函數作為 dict 值的優點包括：

* **提高代碼靈活性**：函數允許您根據需要動態生成值。
* **提高代碼可重用性**：函數可以重複使用來生成不同的值。
* **提高代碼效率**：惰性計算值可以避免不必要的計算。

但是，使用函數作為 dict 值也有一些缺點：

* **使代碼更難理解**：對於不熟悉函數用法的人來說，代碼可能更難理解。
* **可能導致性能問題**：如果函數很昂貴或耗時，則可能會導致性能問題。

這些高級用法可以幫助您更好地利用字典這種強大的數據結構,提高代碼的可讀性和效率。

**for 的高级用法**

1. **並行迭代(Parallel Iteration)**

可以同時遍歷多個可迭代對象。

names = ['Alice', 'Bob', 'Charlie']

ages = [25, 30, 35]

for name, age in zip(names, ages):

print(f"{name} is {age} years old.")

*# Output:*

*# Alice is 25 years old.*

*# Bob is 30 years old.*

*# Charlie is 35 years old.*

1. **enumerate() 函數**

可以獲取元素的索引和值。

fruits = ['apple', 'banana', 'orange']

for i, fruit in enumerate(fruits, start=1):

print(f"{i}. {fruit}")

*# Output:*

*# 1. apple*

*# 2. banana*

*# 3. orange*

1. **嵌套循環(Nested Loops)**

可以使用多個嵌套的 for 循環來遍歷多維數據結構。

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

for row in matrix:

for element in row:

print(element, end=' ')

print()

*# Output:*

*# 1 2 3*

*# 4 5 6*

*# 7 8 9*

1. **迭代字典(Iterating Over Dictionaries)**

可以遍歷字典的鍵、值或鍵值對。

person = {'name': 'Alice', 'age': 25, 'city': 'New York'}

*# Iterate over keys*

for key in person:

print(key)

*# Iterate over values*

for value in person.values():

print(value)

*# Iterate over key-value pairs*

for key, value in person.items():

print(f"{key}: {value}")

1. **else 子句**

for 循環可以與 else 子句一起使用,在循環正常結束時執行。

fruits = ['apple', 'banana', 'orange']

for fruit in fruits:

print(fruit)

else:

print("All fruits have been printed.")

*# Output:*

*# apple*

*# banana*

*# orange*

*# All fruits have been printed.*

1. **break 和 continue 語句**

可以使用 break 語句終止循環,使用 continue 語句跳過當前迭代。

1. **生成器表達式(Generator Expressions)**

可以使用生成器表達式來創建迭代器,從而節省內存。

squares = (x\*\*2 for x in range(10))

for square in squares:

print(square, end=' ')

*# Output: 0 1 4 9 16 25 36 49 64 81*

**while 的高级用法**

1. **無限循環(Infinite Loops)**

當滿足特定條件時,可以使用 break 語句來終止無限循環。

while True:

user\_input = input("Enter 'q' to quit: ")

if user\_input == 'q':

break

print("You entered:", user\_input)

1. **else 子句**

與 for 循環類似, while 循環也可以與 else 子句一起使用,在循環正常結束時執行。

count = 0

while count < 5:

print(count)

count += 1

else:

print("Loop finished!")

1. **使用函數控制循環(Using Functions to Control Loops)**

可以使用函數來控制 while 循環的執行條件,實現更靈活的控制流。

def is\_prime(n):

if n < 2:

return False

for i in range(2, int(n \*\* 0.5) + 1):

if n % i == 0:

return False

return True

n = 2

while is\_prime(n):

print(n)

n += 1

1. **控制循環執行次數(Controlling Loop Iterations)**

可以使用額外的變量來控制 while 循環的執行次數。

count = 0

max\_iterations = 5

while count < max\_iterations:

print(count)

count += 1

1. **生成器函數(Generator Functions)**

可以使用生成器函數來生成無限序列,並在 while 循環中迭代它們。

def fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

fib = fibonacci()

for \_ in range(10):

print(next(fib), end=' ')

1. **併發和異步操作(Concurrency and Asynchronous Operations)**

while 循環常用於實現併發和異步操作,例如在網絡編程和事件驅動編程中。

1. **循環中斷和恢復(Breaking and Resuming Loops)**

可以使用 break 和 continue 語句來控制循環的流程,實現更細粒度的控制。

**python****模擬 goto 用法**

1. **使用標籤和函數**

雖然Python沒有goto語句,但我們可以使用函數來模擬類似的行為。這種方法利用函數的局部變量作為循環標籤。

def func():

print('Started')

label.val = 1

return

label.val = 2

print('Jumped')

label = type('', (), {'val':0})

func()

if label.val == 1:

print('First label')

elif label.val == 2:

print('Second label')

輸出:

Started

First label

1. **使用異常處理**

我們可以利用try/except塊和raise語句來實現類似goto的效果。這種方法雖然不太優雅,但在某些情況下可能是必要的。

try:

print('Started')

raise Exception('Jump')

except Exception as e:

print(e)

print('Jumped')

輸出:

Started

Jump

Jumped

1. **使用生成器**

利用生成器的yield語句,我們可以實現類似goto的功能,但這種方式相對複雜,不太直觀。

def jump():

print('Started')

jump\_position = yield

if jump\_position == 'JUMP':

print('Jumped')

j = jump()

next(j)

j.send('JUMP')

輸出:

Started

Jumped

**4. 使用嵌套 if 語句:**

您可以使用嵌套 if 語句模擬 goto，以便跳到程式碼的特定部分。此方法涉及創建一個嵌套結構的 if 語句，根據特定條件控制執行流程。

def 模擬\_goto(標籤):

if 標籤 == "開始":

# 在 "開始" 標籤處要執行的程式碼

pass

elif 標籤 == "迴圈":

# 在 "迴圈" 標籤處要執行的程式碼

pass

elif 標籤 == "結束":

# 在 "結束" 標籤處要執行的程式碼

pass

# 模擬跳轉到 "開始" 標籤

模擬\_goto("開始")

# 模擬跳轉到 "迴圈" 標籤

模擬\_goto("迴圈")

# 模擬跳轉到 "結束" 標籤

模擬\_goto("結束")

**5. 使用函數和字典:**

另一種方法是使用函數和字典來模擬 goto。此方法涉及創建一個字典來存儲與不同標籤關聯的程式碼塊，並使用函數跳轉到這些程式碼塊。

def goto(標籤):

標籤 = {

"開始": 開始\_程式碼,

"迴圈": 迴圈\_程式碼,

"結束": 結束\_程式碼

}

try:

標籤[標籤]()

except KeyError:

print(f"無效標籤：{標籤}")

def 開始\_程式碼():

# 在 "開始" 標籤處要執行的程式碼

pass

def 迴圈\_程式碼():

# 在 "迴圈" 標籤處要執行的程式碼

pass

def 結束\_程式碼():

# 在 "結束" 標籤處要執行的程式碼

pass

# 模擬跳轉到 "開始" 標籤

goto("開始")

# 模擬跳轉到 "迴圈" 標籤

goto("迴圈")

# 模擬跳轉到 "結束" 標籤

goto("結束")

**set 的高级用法**

1. **集合運算**

Python 支持數學集合的並集、交集、差集和對稱差集等運算。

set1 = {1, 2, 3}

set2 = {2, 3, 4}

*# 並集*

print(set1 | set2) *# {1, 2, 3, 4}*

*# 交集*

print(set1 & set2) *# {2, 3}*

*# 差集*

print(set1 - set2) *# {1}*

print(set2 - set1) *# {4}*

*# 對稱差集*

print(set1 ^ set2) *# {1, 4}*

1. **集合comprehension**

與列表comprehension類似,Python 也支持集合comprehension,用於快速創建集合。

squares = {x\*\*2 for x in range(10)} *# {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}*

1. **集合作為高效的成員檢測工具**

由於集合是無序且不重複的,因此可以將其用作高效的成員檢測工具。

valid\_emails = {'alice@example.com', 'bob@example.com', 'charlie@example.com'}

while True:

email = input("Enter your email: ")

if email in valid\_emails:

print("Valid email")

else:

print("Invalid email")

1. **集合作為高效的去重工具**

利用集合不重複的特性,可以快速從一個序列中移除重複項。

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

unique = set(duplicates) *# {1, 2, 3, 4, 5, 6}*

1. **集合的操作**

Python 還為集合提供了多種實用的操作,如 issubset()、issuperset()、union()、intersection()、difference()等。

set1 = {1, 2, 3}

set2 = {2, 3, 4}

print(set1.issubset(set2)) *# False*

print(set1.issuperset(set2)) *# False*

print(set1.union(set2)) *# {1, 2, 3, 4}*

print(set1.intersection(set2)) *# {2, 3}*

print(set1.difference(set2)) *# {1}*

1. **不可變集合(frozenset)**

Python 還提供了 frozenset，它是不可變的集合。frozenset 可以作為字典的鍵或其他集合的元素。

frozenset2 = frozenset({3, 4, 5})

dict1 = {frozenset1: 'abc', frozenset2: 'def'}

print(dict1) *# {frozenset({1, 2, 3}): 'abc', frozenset({3, 4, 5}): 'def'}*

**python模擬 switch/case 用法**

1. **使用字典映射**

這是模擬 switch/case 最直接的方式,通過將 case 值作為字典的鍵,將對應的操作作為值。

def switch(operator, a, b):

operations = {

'+': a + b,

'-': a - b,

'\*': a \* b,

'/': a / b

}

return operations.get(operator, 'Invalid operation')

result = switch('+', 2, 3)

print(result) *# 輸出: 5*

1. **使用單分派泛函數**

Python 的函數可以通過內置的 singledispatch 裝飾器實現單分派泛函數,根據參數的類型選擇執行不同的函數體。

from functools import singledispatch

@singledispatch

def operation(operator, a, b):

print('Invalid operation')

@operation.register(int)

def \_(operator, a, b):

operations = {

'+': a + b,

'-': a - b,

'\*': a \* b,

'/': a / b

}

return operations.get(operator, 'Invalid operation')

@operation.register(str)

def \_(operator, a, b):

if operator == 'concat':

return a + b

else:

return 'Invalid operation for strings'

result = operation('+', 2, 3)

print(result) *# 輸出: 5*

result = operation('concat', 'Hello', 'World')

print(result) *# 輸出: HelloWorld*

1. **使用類和多重繼承**

通過創建多個子類繼承同一個父類,然後在父類中根據子類的類型調用不同的方法。

class Operation:

def eval(self, a, b):

pass

class Addition(Operation):

def eval(self, a, b):

return a + b

class Subtraction(Operation):

def eval(self, a, b):

return a - b

def switch(operation, a, b):

operations = {

'+': Addition(),

'-': Subtraction()

}

return operations.get(operation, None).eval(a, b)

result = switch('+', 2, 3)

print(result) *# 輸出: 5*

result = switch('-', 5, 3)

print(result) *# 輸出: 2*

1. **使用 lambda 函數和字典映射**

通過將操作函數作為字典值存儲,根據 case 調用對應的 lambda 函數。

operations = {

'+': lambda a, b: a + b,

'-': lambda a, b: a - b,

'\*': lambda a, b: a \* b,

'/': lambda a, b: a / b

}

def switch(operator, a, b):

return operations.get(operator, lambda a, b: 'Invalid operation')(a, b)

result = switch('+', 2, 3)

print(result) *# 輸出: 5*

**函式 的高级用法(1)**

1. **高階函數(Higher-Order Functions)**

Python 中的函數可以作為參數傳遞給另一個函數,或者從函數返回另一個函數。這就是所謂的高階函數。

*# 將函數作為參數傳遞*

def apply\_operation(func, x, y):

return func(x, y)

def add(a, b):

return a + b

def multiply(a, b):

return a \* b

print(apply\_operation(add, 2, 3)) *# 輸出: 5*

print(apply\_operation(multiply, 2, 3))*# 輸出: 6*

*# 從函數返回另一個函數*

def create\_multiplier(n):

def multiplier(x):

return x \* n

return multiplier

double = create\_multiplier(2)

triple = create\_multiplier(3)

print(double(5)) *# 輸出: 10*

print(triple(5)) *# 輸出: 15*

1. **Lambda 函數(Lambda Functions)**

Lambda 函數是一種小型匿名函數,常用於簡單的單行函數定義。它們通常與高階函數一起使用。

*# 使用 lambda 函數作為參數*

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

doubled = list(map(lambda x: x \* 2, numbers))

print(doubled) *# 輸出: [2, 4, 6, 8, 10]*

*# 使用 lambda 函數進行排序*

students = [('Alice', 25), ('Bob', 30), ('Charlie', 20)]

sorted\_students = sorted(students, key=lambda x: x[1])

print(sorted\_students) *# 輸出: [('Charlie', 20), ('Alice', 25), ('Bob', 30)]*

1. **閉包(Closures)**

閉包是一種函數,它記住了自己被定義時的環境。這使得它可以訪問外部函數中的變量。

def outer\_function(x):

y = 4

def inner\_function(z):

print(f"x = {x}, y = {y}, z = {z}")

return x + y + z

return inner\_function

func = outer\_function(2)

result = func(3) *# 輸出: x = 2, y = 4, z = 3*

print(result) *# 輸出: 9*

1. **裝飾器(Decorators)**

裝飾器是一種特殊的閉包,用於修改函數的行為。它們可以在不更改原始函數源代碼的情況下擴展其功能。

def log\_args(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling {func.\_\_name\_\_} with args={args} and kwargs={kwargs}")

return func(\*args, \*\*kwargs)

return wrapper

@log\_args

def add(x, y):

return x + y

result = add(2, 3)

*# 輸出: Calling add with args=(2, 3) and kwargs={}*

*# 輸出: 5*

1. **生成器函數(Generator Functions)**

生成器函數是一種特殊的函數,可以暫停和恢復它們的執行狀態,從而實現迭代器協議。

def count\_up\_to(n):

i = 0

while i < n:

yield i

i += 1

counter = count\_up\_to(5)

print(list(counter)) *# 輸出: [0, 1, 2, 3, 4]*

1. **遞歸函數(Recursive Functions)**

Python 支持函數自身調用自身,這就是遞歸。適當使用可以使代碼更加簡潔和易於理解。

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) *# 輸出: 120*

1. **函數註釋和類型提示**

Python 3 引入了函數註釋和類型提示,可以為函數參數和返回值添加類型信息,提高代碼的可讀性和維護性。

def greet(name: str) -> str:

"""

Greets the given name.

Args:

name (str): The name to greet.

Returns:

str: The greeting message.

"""

return f"Hello, {name}!"

print(greet("Alice")) *# 輸出: Hello, Alice!*

**函式 的高级用法(2)**

Python 函式（function）是一種基本的編程結構，可以將代碼分組為可重複使用的模組。Python 提供了許多高級功能來擴展函數的功能，以下是一些常見的高級用法：

1.函數參數

函數參數是指傳遞給函數的值，可以通過關鍵字參數或者位置參數來傳遞。關鍵字參數是通過名稱來傳遞參數，而位置參數則是按照順序傳遞參數。

範例：

def greet(name):

print("Hello, " + name)

greet("John") # Output: Hello, John

2.默認參數

默認參數是指在函數呼叫時未提供參數時，函數會使用默認值。這樣可以方便地呼叫函數，並且不需要每次都提供所有參數。

範例：

def greet(name="World"):

print("Hello, " + name)

greet() # Output: Hello, World

3.可變參數

可變參數是指函數可以接受任意數量的參數。這樣可以方便地處理不同數量的參數，並且可以使函數更加靈活。

範例：

def sum(\*numbers):

total = 0

for number in numbers:

total += number

return total

print(sum(1, 2, 3)) # Output: 6

4.返回值

返回值是指函數執行後返回的值。返回值可以是任意類型的值，包括數據、物件、函數等。

範例：

def add(a, b):

return a + b

result = add(1, 2)

print(result) # Output: 3

5.匿名函數

匿名函數是指沒有名稱的函數，可以直接創建並使用。這樣可以方便地創建臨時函數，並且可以使代碼更加簡潔。

範例：

func = lambda x: x \* 2

print(func(5)) # Output: 10

6.裝飾器

裝飾器是指用於修改函數行為的函數。裝飾器可以在函數執行前或執行後進行操作，從而實現對函數的控制。

範例：

def decorator(func):

def wrapper():

print("Before function call")

func()

print("After function call")

return wrapper

@decorator

def greet():

print("Hello, World!")

greet()

# Output:

# Before function call

# Hello, World!

# After function call

7.閉包

閉包是指引用了自由變量的函數。自由變量是指在函數外部定義的變量，閉包可以在函數內部訪問這些變量。閉包可以用於實現狀態保持和共用數據等功能。

範例：

def outer\_function():

x = 10

def inner\_function():

nonlocal x

x += 1

print(x)

return inner\_function

inner\_function = outer\_function()

inner\_function() # Output: 11

8.生成器

生成器是指可以一次生成多個值的函數。生成器可以用於實現反覆運算器、反覆運算器鏈等功能。

範例：

def fibonacci(n):

a, b = 0, 1

for \_ in range(n):

yield a

a, b = b, a+b

for num in fibonacci(10):

print(num)

# Output:

# 0

# 1

# 1

# 2

# 3

# 5

# 8

# 13

# 21

# 34

9.異常處理

異常處理是指當函數執行發生錯誤時，可以使用異常處理機制來處理錯誤。異常處理可以使代碼更加健壯，並且可以方便地排除錯誤。

範例：

try:

result = 1 / 0

except ZeroDivisionError as e:

print("Error:", e)

else:

print("Result:", result)

finally:

print("Finished")

# Output:

# Error: division by zero

# Finished

10.文檔注釋

文檔注釋是指在函數中添加的注釋，用於說明函數的功能和使用方法。文檔注釋可以使代碼更加易讀，並且可以方便地查看函數的資訊。

範例：

def greet(name):

"""

打招呼函數

:param name: 名稱

:return: None

"""

print("Hello, " + name)

greet("John") # Output: Hello, John

**Python 內建物件**

Python 內建物件是基本資料型態和函式，可以直接在 Python 解譯器或程式碼中使用，無需匯入任何其他模組。它們為構建更複雜的程式提供了基礎，對於理解語言的核心功能至關重要。

**內建物件的種類**

Python 內建物件可廣泛分為以下類別：

1. **數字**：這些物件代表數值值，包括整數、浮點數和複數。它們支援加、減、乘、除和比較等運算。

範例：10, 3.14, 1j

1. **字串**：這些物件代表字元序列，通常用引號括起來。它們支援連接、切片、索引和搜尋等運算。

範例："Hello, World!", 'Python Programming', r"C:\Users\username\myfile.txt"

1. **列表**：這些物件代表有序的項目集合，用方括號括起來。它們支援動態新增、刪除和修改元素。

範例：[1, 2, 3, 4], ["apple", "banana", "cherry"], [[1, 2], [3, 4], [5, 6]]

1. **元組**：這些物件代表不可變的有序項目集合，用括號括起來。創建後無法修改。

範例：(1, 2, 3), ("John", "Doe", 30), ((1, 2), (3, 4))

1. **集合**：這些物件代表無序的唯一項目集合，用大括號括起來。它們支援聯集、交集和差集等運算。

範例：{1, 2, 3, 4}, {"apple", "banana", "cherry"}, {(1, 2), (3, 4)}

1. **字典**：這些物件代表無序的鍵值對集合，用大括號括起來。它們在鍵和值之間提供映射。

範例：{"name": "Alice", "age": 30, "city": "New York"}, {"fruits": ["apple", "banana", "cherry"], "numbers": [1, 2, 3]}

1. **布林**：這些物件代表邏輯值，True 或 False。它們用於條件陳述式和布林表達式。

範例：True, False

1. **None**：此物件代表值不存在。它通常用作佔位符或指示變數未分配值。

範例：None

**存取內建物件**

內建物件可以直接在 Python 解譯器或程式碼中存取，無需明確匯入。您可以使用它們來執行各種操作並創建更複雜的資料結構。

**內建物件使用範例**

1. **算術運算**：

# 加法

result = 5 + 3

print(result) # 輸出：8

# 減法

difference = 10 - 4

print(difference) # 輸出：6

# 乘法

product = 2 \* 7

print(product) # 輸出：14

# 除法

quotient = 16 / 4

print(quotient) # 輸出：4.0

1. **字串操作**：

# 連接

greeting = "Hello" + " " + "World!"

print(greeting) # 輸出：Hello World!

# 切片

extracted\_string = "Python Programming"[5:17]

print(extracted\_string) # 輸出：on Programming

# 索引

first\_character = "Welcome"[0]

print(first\_character) # 輸出：W

# 搜尋

substring\_index = "Python is fun!".find("fun")

print(substring\_index) # 輸出：10

1. **列表操作**：

# 新增元素

numbers = [1, 2, 3]

numbers.append(4)

print(numbers) # 輸出：[1, 2, 3, 4]

share

more\_vert

**4. 元組操作**

元組是不可變的資料結構，因此不能像列表那樣新增、刪除或修改元素。但是，您可以使用索引和切片來存取元組中的元素。

# 創建元組

fruits = ("apple", "banana", "cherry")

# 索引

first\_fruit = fruits[0]

print(first\_fruit) # 輸出：apple

# 切片

tropical\_fruits = fruits[1:3]

print(tropical\_fruits) # 輸出：('banana', 'cherry')

**5. 集合操作**

集合是無序的唯一項目集合。它們支援聯集、交集、差集等運算。

# 創建集合

set1 = {1, 2, 3}

set2 = {3, 4, 5}

# 聯集

union = set1 | set2

print(union) # 輸出：{1, 2, 3, 4, 5}

# 交集

intersection = set1 & set2

print(intersection) # 輸出：{3}

# 差集

difference = set1 - set2

print(difference) # 輸出：{1, 2}

**6. 字典操作**

字典是無序的鍵值對集合。您可以使用鍵來存取字典中的值。

# 創建字典

student = {"name": "Alice", "age": 30, "city": "New York"}

# 存取值

name = student["name"]

print(name) # 輸出：Alice

# 更新值

student["age"] = 31

print(student) # 輸出：{'name': 'Alice', 'age': 31, 'city': 'New York'}

# 檢查鍵是否存在

if "city" in student:

print("city key exists") # 輸出：city key exists

**7. 布林運算**

布林運算用於組合布林值 (True 或 False)。

# 邏輯與

result = 10 > 5 and 3 == 2

print(result) # 輸出：False

# 邏輯或

result = 10 > 5 or 3 == 2

print(result) # 輸出：True

# 非

result = not (10 > 5)

print(result) # 輸出：False

**8. None 物件**

None 物件代表值不存在。它通常用作佔位符或指示變數未分配值。

# 檢查變數是否未定義

age = None

if age is None:

print("age is not defined") # 輸出：age is not defined

# 將變數設為 None

name = "Alice"

name = None

print(name) # 輸出：None

**\_\_init\_\_ ()與 \_\_new\_\_ ()區別**

在 Python 中，\_\_init\_\_() 和 \_\_new\_\_() 都是特殊方法（special method），它們在創建和初始化類別物件時扮演著重要角色。然而，它們之間存在著關鍵的區別。

**1. \_\_init\_\_() 方法**

\_\_init\_\_() 方法通常稱為 **初始化方法** 或 **建構子**（constructor）。它在創建類別物件時自動調用。\_\_init\_\_() 方法負責初始化物件的屬性和狀態。

舉例來說，假設我們有一個表示人的類別 Person：

class Person:

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

self.name = name

self.age = age

在這個例子中，\_\_init\_\_() 方法接受兩個參數：name 和 age。它將這些參數分配給物件的屬性 self.name 和 self.age。

**2. \_\_new\_\_() 方法**

\_\_new\_\_() 方法通常稱為 **新物件方法**。它在創建類別物件之前調用。\_\_new\_\_() 方法負責實際創建物件。

\_\_new\_\_() 方法必須返回一個物件。如果它返回 None，則物件創建失敗。

在某些情況下，您可能需要自訂 \_\_new\_\_() 方法。例如，您可以使用它來實現單例（singleton）模式或控制物件創建的邏輯。

**\_\_init\_\_() 和 \_\_new\_\_() 之間的區別**

以下是一些關鍵區別：

* **調用時機：** \_\_init\_\_() 在物件創建後調用，而 \_\_new\_\_() 在物件創建之前調用。
* **目的：** \_\_init\_\_() 負責初始化物件的屬性和狀態，而 \_\_new\_\_() 負責實際創建物件。
* **返回值：** \_\_init\_\_() 沒有返回值，而 \_\_new\_\_() 必須返回一個物件。

**總結**

\_\_init\_\_() 和 \_\_new\_\_() 都是 Python 中強大的工具，可讓您控制類別物件的創建和初始化。瞭解它們之間的區別對於編寫健壯且可維護的程式碼至關重要。

以下是一些額外的注意事項：

* 在大多數情況下，您無需自訂 \_\_new\_\_() 方法。\_\_init\_\_() 通常足以滿足初始化物件的需求。
* 如果您確實需要自訂 \_\_new\_\_() 方法，請務必小心謹慎。錯誤的實現可能會導致意外的行為或物件創建失敗。

**變量查找機制**

在 Python 中，變量查找機制是確定變量值的規則。當您在程式碼中使用變量時，Python 解釋器會遵循特定的步驟來查找該變量的值。

**作用域**

作用域是變量可見的範圍。Python 有四種作用域：

1. **局部作用域**：局部作用域是指函式或 lambda 表達式的內部。在局部作用域中，變量只能在該函式或 lambda 表達式內部使用。

def my\_function():

age = 30

print(age) # 輸出：30

my\_function()

在這個例子中，age 變量在 my\_function() 函式的局部作用域中定義。在函式外部，無法訪問該變量。

1. **全域作用域**：全域作用域是指整個程式碼的範圍。在全域作用域中，變量可以在程式碼的任何地方使用。

age = 25

def my\_function():

print(age) # 輸出：25

my\_function()

在這個例子中，age 變量在全域作用域中定義。它可以在 my\_function() 函式內部和程式碼的外部使用。

1. **嵌套作用域**：嵌套作用域是指在另一個作用域內的作用域。在嵌套作用域中，內部作用域的變量可以訪問外部作用域的變量。

def outer\_function():

age = 40

def inner\_function():

print(age) # 輸出：40

inner\_function()

outer\_function()

在這個例子中，age 變量在 outer\_function() 函式的局部作用域中定義。inner\_function() 函式位於 outer\_function() 函式的嵌套作用域中，因此它可以訪問 outer\_function() 函式的 age 變量。

1. **內建作用域**：內建作用域是指內建函式和變量的作用域。內建作用域在 Python 程式碼的任何地方都可用。

print("Hello, World!") # 使用內建的 print() 函式

在這個例子中，print() 函式是內建函式。它可以在程式碼的任何地方使用，無需匯入任何模組。

**變量查找順序**

當您在程式碼中使用變量時，Python 解釋器會按照以下順序查找該變量的值：

1. **局部作用域**：Python 解釋器首先會在當前函式或 lambda 表達式的局部作用域中查找變量。
2. **嵌套作用域**：如果在局部作用域中找不到變量，Python 解釋器會在嵌套作用域中查找變量。
3. **全域作用域**：如果在嵌套作用域中找不到變量，Python 解釋器會在全域作用域中查找變量。
4. **內建作用域**：如果在全域作用域中找不到變量，Python 解釋器會在內建作用域中查找變量。

如果在任何作用域中都找不到變量，則會發生 NameError 異常。

**理解變量查找機制的重要性**

理解變量查找機制對於編寫健壯且易於理解的 Python 程式碼至關重要。它可以幫助您避免變量命名衝突和 NameError 異常。

**MRO（方法解析順序）與多繼承**

**MRO（Method Resolution Order）**

在 Python 中，MRO（Method Resolution Order）是指方法解析順序。當您使用多繼承時，MRO 決定瞭解析器在類別中查找方法時遵循的順序。

多繼承允許一個類別繼承自多個父類別。這可以使您重用程式碼並從多個來源獲取功能。但是，它也可能導致歧義，因為多個父類別可能定義具有相同名稱的方法。

MRO 解決了這個問題。它定義瞭解析器在類別中查找方法時遵循的順序。這確保瞭解析器始終找到預期的方法。

**MRO 的計算**

MRO 的計算取決於類別的繼承結構。一般來說，MRO 包含以下元素：

* 當前類別本身
* 所有直接父類別（從左到右）
* 遞歸地，每個父類別的 MRO

MRO 的計算遵循以下步驟：

1. 創建一個空列表 mro。
2. 將當前類別添加到 mro 的末尾。
3. 對於每個直接父類別：
   * 如果父類別的 MRO 不在 mro 中，請將其添加到 mro 的末尾。
   * 否則，跳過此父類別。
4. 遞歸地，對於每個父類別的 MRO，重複步驟 3。

**MRO 的示例**

以下是一個簡單的多繼承示例：

class A:

def method1(self):

print("A.method1()")

class B:

def method1(self):

print("B.method1()")

class C(A, B):

pass

在此示例中，C 類別繼承自 A 類別和 B 類別。如果我們在 C 類別的實例上調用 method1() 方法，解析器將遵循以下 MRO 來查找該方法：

[C, A, B]

這意味著：

1. 解析器首先檢查 C 類別是否定義了 method1() 方法。
2. 如果 C 類別沒有定義 method1() 方法，則解析器將檢查 A 類別。
3. 如果 A 類別也沒有定義 method1() 方法，則解析器將檢查 B 類別。

在此示例中，B 類別定義了 method1() 方法，因此該方法將被調用。

**MRO 的重要性**

MRO 對於理解 Python 中多繼承的工作原理至關重要。它可以幫助您避免方法名稱衝突並預測解析器將如何查找方法。

以下是一些有關 MRO 的提示：

* 使用描述性方法名稱來提高程式碼的可讀性和可維護性。
* 了解 MRO 的計算方式，以便更好地理解多繼承的行為。
* 使用 super() 函式來明確指定您要調用的方法的父類別。

**描述符機制（descriptor protocol）**

描述符機制（descriptor protocol）是 Python 中一種強大的工具，可讓您自定義物件屬性和方法的行為。它允許您以一致和可預測的方式在不同物件上實現通用行為。

**描述符的種類**

描述符可以分為以下三種類別：

1. **資料描述符**：資料描述符控制物件屬性的存取。它們實現了 \_\_get\_\_()、\_\_set\_\_() 和 \_\_delete\_\_() 方法之一或全部。
   * \_\_get\_\_() 方法用於獲取屬性值。
   * \_\_set\_\_() 方法用於設定屬性值。
   * \_\_delete\_\_() 方法用於刪除屬性。
2. **非資料描述符**：非資料描述符不控制物件屬性的存取。它們僅實現了 \_\_get\_\_() 方法。
   * \_\_get\_\_() 方法用於獲取屬性值。
3. **類描述符**：類描述符控制類別屬性的存取。它們在類別上調用時而不是在類別實例上調用。它們實現了 \_\_get\_\_()、\_\_set\_\_() 和 \_\_delete\_\_() 方法之一或全部，但方法名稱會加上 \_\_class\_\_ 後綴。

**描述符的示例**

以下是一個簡單的資料描述符示例：

class AgeDescriptor:

def \_\_init\_\_(self, initial\_age):

self.age = initial\_age

def \_\_get\_\_(self, instance, owner):

print("Getting age...")

return self.age

def \_\_set\_\_(self, instance, value):

print("Setting age...")

self.age = value

class Person:

age = AgeDescriptor(25)

person = Person()

print(person.age) # 輸出：Getting age...25

person.age = 30

print(person.age) # 輸出：Setting age...30

在此示例中，AgeDescriptor 類別是一個資料描述符。它實現了 \_\_get\_\_() 和 \_\_set\_\_() 方法，以自定義物件 age 屬性的存取。

**描述符的用途**

描述符可以用於多種目的，包括：

* 實現屬性驗證和驗證
* 實現屬性緩存
* 實現代理（proxy）行為
* 實現元編程（metaprogramming）

**描述符的注意事項**

使用描述符時，請注意以下事項：

* 描述符可能使程式碼更難理解和維護。
* 描述符可能會導致意外的副作用。
* 應謹慎使用描述符。

**區別 \_\_getattr\_\_ ()與 \_\_getattribute\_\_ ()**

在Python中,\_\_getattr\_\_和\_\_getattribute\_\_都是用於獲取屬性值的特殊方法,但它們的作用和觸發時機是不同的。

1. \_\_getattr\_\_方法:
   * \_\_getattr\_\_是一個物件方法,當訪問一個不存在的屬性時被調用。
   * 只有在不存在所訪問的屬性時,Python才會嘗試調用\_\_getattr\_\_方法。
   * 通常用於實現延遲加載或代理屬性等功能。

範例:

class Person:

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

self.name = name

def \_\_getattr\_\_(self, attr):

print(f"嘗試訪問不存在的屬性: {attr}")

return None

p = Person("John")

print(p.name) *# 輸出: John*

print(p.age) *# 嘗試訪問不存在的屬性: age, 輸出: None*

1. \_\_getattribute\_\_方法:
   * \_\_getattribute\_\_也是一個物件方法,但它會在獲取任何屬性時被調用,無論屬性是否存在。
   * 這個方法可以用於實現屬性訪問的統一控制或記錄。
   * 如果在\_\_getattribute\_\_方法中直接訪問其他屬性,會導致無限遞迴,因此需要使用super().\_\_getattribute\_\_來獲取屬性值。

範例:

class Person:

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

self.name = name

def \_\_getattribute\_\_(self, attr):

print(f"獲取屬性: {attr}")

return super().\_\_getattribute\_\_(attr)

p = Person("John")

print(p.name) *# 獲取屬性: name, 輸出: John*

print(p.age) *# 獲取屬性: age, 輸出: AttributeError: 'Person' object has no attribute 'age'*

以下是一些關鍵區別：

* **調用時機：** \_\_getattr\_\_() 僅在物件中找不到屬性時調用，而 \_\_getattribute\_\_() 始終在存取物件屬性時調用。
* **目的：** \_\_getattr\_\_() 負責處理未定義屬性的訪問，而 \_\_getattribute\_\_() 負責實現物件屬性的實際存取。
* **參數：** \_\_getattr\_\_() 接收一個參數（屬性名稱），而 \_\_getattribute\_\_() 接收兩個參數（物件本身和屬性名稱）。
* **返回值：** \_\_getattr\_\_() 必須返回一個值。如果它返回 None，則會拋出 AttributeError 異常。\_\_getattribute\_\_() 必須返回屬性值。

總結:

* \_\_getattr\_\_用於處理不存在的屬性訪問,可以返回替代值或引發異常。
* \_\_getattribute\_\_用於控制所有屬性訪問的行為,可以實現屬性訪問的統一管理或記錄。
* 如果同時定義了這兩個方法,\_\_getattribute\_\_會先被調用。
* 在\_\_getattribute\_\_方法中訪問屬性時,需要使用super().\_\_getattribute\_\_來避免無限遞迴。

**property 裝飾器**

property 裝飾器是一種 Python 工具，可讓您將函式變為物件屬性。它允許您以更直觀和 Pythonic 的方式存取和設定屬性值。

**基本用法**

要使用 property 裝飾器，請在要變為屬性的函式之前添加 @property 裝飾器。例如：

class Person:

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

self.name = name

self.\_age = age # 使用下劃線開頭表示私有屬性

@property

def age(self):

return self.\_age

@age.setter

def age(self, value):

if value < 0:

raise ValueError("Age must be non-negative")

self.\_age = value

在此示例中，age 函式被裝飾為 property。這意味著它將成為 Person 類別的屬性。您可以使用 person.age 獲取或設定屬性值，就像它是一個普通的屬性一樣。

**property 的優點**

使用 property 裝飾器具有以下優點：

* **更直觀：** 您可以使用點號語法（.age）存取和設定屬性值，就像它們是普通的屬性一樣。
* **更 Pythonic：** 它允許您以更 Pythonic 的方式定義和使用屬性。
* **更靈活：** 您可以自訂屬性值的獲取和設定行為。

**進階用法**

property 裝飾器允許您自訂屬性值的獲取和設定行為。您可以使用 @property.getter 和 @property.setter 裝飾器來分別定義獲取器和設定器方法。例如：

class Person:

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

self.name = name

self.\_age = age

@property

def age(self):

return self.\_age

@age.getter

def age(self):

print("Getting age...")

return self.\_age

@age.setter

def age(self, value):

if value < 0:

raise ValueError("Age must be non-negative")

print("Setting age...")

self.\_age = value

在此示例中，age 函式被裝飾為 property。它還具有 @property.getter 和 @property.setter 裝飾器，用於自訂獲取器和設定器方法。這允許您在獲取或設定屬性值時執行自訂邏輯。

**property 的注意事項**

使用 property 裝飾器時，請注意以下事項：

* property 裝飾器不能用於靜態方法或類方法。
* 應謹慎使用 property 裝飾器，因為它們可能會使程式碼更難理解和維護。

**元類（metaclass）**

在 Python 中，元類是一種特殊類別，用於創建其他類別。它們提供了對類創建過程的更多控制和靈活性。

**理解元類**

要理解元類，首先要瞭解類別的創建過程。當您在 Python 中定義類別時，會發生以下情況：

1. Python 解釋器解析類別定義。
2. 解釋器創建一個新類別物件。
3. 解釋器將類別的屬性和方法添加到新類別物件中。
4. 返回新類別物件。

元類介入瞭類創建過程的第二步。當您使用元類創建類別時，元類會被用於創建新類別物件。這允許元類自訂類別的屬性和方法。

**元類的使用案例**

元類可以用於多種目的，包括：

* **實現自動化屬性：** 元類可以用於自動創建 getter 和 setter 方法，從而簡化屬性定義。
* **實現單例模式：** 元類可以用於確保類別只創建一個實例。
* **實現驗證：** 元類可以用於驗證類別屬性和方法的有效性。
* **實現擴展：** 元類可以用於為類別添加新功能。

Python中默認的元類是type。當你定義一個新類時,Python會自動調用type元類來創建該類對象。你也可以定義自己的元類,並讓新創建的類使用你自定義的元類。

以下是一個使用元類的基本示例:

*# 定義元類*

class MyMeta(type):

def \_\_new\_\_(cls, name, bases, attrs):

print(f"創建新類: {name}")

return super().\_\_new\_\_(cls, name, bases, attrs)

*# 使用自定義元類*

class MyClass(metaclass=MyMeta):

pass

*# 輸出: 創建新類: MyClass*

在這個例子中:

1. 我們定義了一個名為MyMeta的元類,繼承自type。
2. 在\_\_new\_\_方法中,我們列印出要創建的類的名稱。
3. 我們使用metaclass=MyMeta參數來指定MyClass使用我們自定義的MyMeta元類。
4. 當創建MyClass時,Python會先調用MyMeta的\_\_new\_\_方法,然後再創建MyClass對象。

元類的一個常見用途是在類定義時對類屬性進行處理或修改。下面是一個在類定義時自動將方法加上裝飾器的示例:

def debug(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling {func.\_\_name\_\_}")

return func(\*args, \*\*kwargs)

return wrapper

class MetaDebug(type):

def \_\_new\_\_(cls, name, bases, attrs):

for key, value in attrs.items():

if callable(value):

attrs[key] = debug(value)

return super().\_\_new\_\_(cls, name, bases, attrs)

class MyClass(metaclass=MetaDebug):

def foo(self):

pass

def bar(self):

pass

obj = MyClass()

obj.foo() *# 輸出: Calling foo*

obj.bar() *# 輸出: Calling bar*

在這個例子中:

1. 我們定義了一個debug裝飾器,用於列印方法調用的資訊。
2. 我們定義了一個MetaDebug元類,在\_\_new\_\_方法中遍歷類屬性,如果屬性是一個可調用對象(函數或方法),就用debug裝飾器包裹它。
3. 我們讓MyClass使用MetaDebug元類,這樣在創建MyClass時,所有的方法都會自動被debug裝飾器包裹。

元類是Python中一個很高級的特性,它提供了一種控制和定製類創建過程的方式。使用元類可以實現一些高級技術,如自動應用裝飾器、自動註冊類、單例模式等。但是,由於元類涉及到Python內部的類創建機制,因此它也是一個相對複雜的概念,需要深入理解Python的對象模型。

**物件協定（Object Protocol）**

在Python中,對象協議(Object Protocol)是一組特殊的方法,它們定義了對象應該如何在特定情況下表現。這些特殊方法可以被覆寫,從而定製對象的行為。遵循對象協議可以使Python對象能夠與內置函數和操作符更好地協作。

物件協定涵蓋了各種主題，包括：

* **屬性訪問：** 物件如何暴露其屬性以及如何獲取和設置屬性值。
* **方法調用：** 物件如何回應方法調用以及如何傳遞參數和返回值。
* **比較：** 物件如何與其他物件進行比較以及如何確定它們是否相等。
* **雜湊：** 物件如何被雜湊以及如何生成雜湊值。
* **字串表示：** 物件如何轉換為字串表示形式。

**物件協定的重要性**

物件協定對於 Python 的運行至關重要。它確保了所有物件都以一致和可預測的方式進行交互。這使得 Python 程式師可以輕鬆地編寫可重用和可維護的代碼。

**物件協定的組成部分**

物件協定由以下部分組成：

* **特殊方法（Special Methods）：** 特殊方法是用雙底線（\_\_）開頭的特殊方法。它們用於實現物件協定的各種方面，例如屬性訪問、方法調用、比較等。
* **描述符（Descriptors）：** 描述符是用於自訂物件屬性和方法行為的特殊物件。它們允許您以更靈活的方式控制物件的行為。
* **元類（Metaclass）：** 元類是用於創建其他類的特殊類。它們允許您控制類創建過程並為類添加新功能。

以下是一些常見的Python對象協議方法:

1. **構造和表示**

* \_\_new\_\_(cls, ...)：用於創建新實例對象。
* \_\_init\_\_(self, ...)：用於初始化實例對象。
* \_\_repr\_\_(self)：用於生成對象的字串表示形式,主要用於調試和日誌。
* \_\_str\_\_(self)：用於生成對象的可讀字串表示形式。

1. **容器類型**

* \_\_len\_\_(self)：實現len(obj)。
* \_\_getitem\_\_(self, key)：實現自定義對象的索引訪問，如obj[key]。
* \_\_setitem\_\_(self, key, value)：實現自定義對象的索引賦值，如obj[key] = value。
* \_\_delitem\_\_(self, key)：實現自定義對象的索引刪除，如del obj[key]。

1. **可反覆運算對象**

* \_\_iter\_\_(self)：返回一個反覆運算器對象。
* \_\_next\_\_(self)：從反覆運算器中獲取下一個值。

1. **上下文管理器**

* \_\_enter\_\_(self)：進入with語句的運行時上下文相關的設置。
* \_\_exit\_\_(self, exc\_type, exc\_value, traceback)：退出運行時上下文相關的釋放資源。

1. **算術運算**

* \_\_add\_\_(self, other)：實現self + other。
* \_\_sub\_\_(self, other)：實現self - other。
* \_\_mul\_\_(self, other)：實現self \* other。
* \_\_truediv\_\_(self, other)：實現self / other。

1. **比較運算**

* \_\_lt\_\_(self, other)：實現self < other。
* \_\_le\_\_(self, other)：實現self <= other。
* \_\_eq\_\_(self, other)：實現self == other。
* \_\_ne\_\_(self, other)：實現self != other。

這些只是對象協議中的一部分方法。通過實現這些特殊方法,我們可以定製對象的行為,使其與Python的內置功能和語法更加協調。

以下是一個實現自定義序列類型的示例:

class CustomSequence:

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

self.\_data = data

def \_\_len\_\_(self):

return len(self.\_data)

def \_\_getitem\_\_(self, index):

return self.\_data[index]

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

return f"CustomSequence({self.\_data})"

seq = CustomSequence([1, 2, 3, 4, 5])

print(len(seq)) *# 5*

print(seq[2]) *# 3*

print(repr(seq)) *# CustomSequence([1, 2, 3, 4, 5])*

在這個示例中,我們定義了一個CustomSequence類,它實現了\_\_len\_\_、\_\_getitem\_\_和\_\_repr\_\_方法。因此,該類的實例可以像序列一樣使用len()函數獲取長度,使用索引訪問元素,並且在列印時有一個可讀的字串表示形式。

**操作符重載(Operator Overloading)**

在Python中,我們可以透過操作符重載(Operator Overloading)來實現中綴語法(Infix Notation)。中綴語法是一種將運算元置於運算符號之間的表示方式,例如2 + 3。Python允許我們為自定義類重載大多數的運算符,使得自定義對象可以像內置類型一樣參與各種運算。

要實現中綴語法,我們需要在自定義類中實現相應的特殊方法。以下是一個實現向量加法的例子:

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

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

return f"Vector({self.x}, {self.y})"

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

x = self.x + other.x

y = self.y + other.y

return Vector(x, y)

v1 = Vector(2, 3)

v2 = Vector(5, 7)

v3 = v1 + v2 *# 使用中綴語法進行向量加法*

print(v3) *# 輸出: Vector(7, 10)*

在這個例子中:

1. 我們定義了一個Vector類,用於表示二維向量。
2. 在\_\_repr\_\_方法中,我們提供了一種可讀的向量表示方式。
3. 我們實現了\_\_add\_\_方法,這個方法接受另一個向量作為參數,並返回一個新的向量,表示兩個向量相加的結果。
4. 在創建v1和v2兩個向量後,我們使用v1 + v2的中綴語法進行向量加法運算,結果存儲在v3中。

Python允許我們為大多數的二元運算符重載特殊方法,包括\_\_add\_\_(+)、\_\_sub\_\_(-)、\_\_mul\_\_(\*)、\_\_truediv\_\_(/)等。我們還可以為一元運算符如\_\_neg\_\_(-)和\_\_pos\_\_(+)實現重載方法。

除了數學運算外,我們還可以重載其他運算符,如比較運算符(\_\_lt\_\_、\_\_le\_\_、\_\_eq\_\_等)、索引運算符(\_\_getitem\_\_)等,從而使自定義對象的行為更加類似於內置類型。

以下是一個實現可索引對象的例子:

class CustomList:

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

self.data = data

def \_\_len\_\_(self):

return len(self.data)

def \_\_getitem\_\_(self, index):

return self.data[index]

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

print(len(my\_list)) *# 輸出: 5*

print(my\_list[2]) *# 輸出: 3*

在這個例子中,我們定義了一個CustomList類,並實現了\_\_len\_\_和\_\_getitem\_\_方法。因此,CustomList的實例可以像列表一樣使用len()函數獲取長度,並使用索引來訪問元素。

透過操作符重載,我們可以為自定義對象提供熟悉的語法和行為,使代碼更加簡潔、可讀性更高。但是,過度使用操作符重載可能會影響代碼的可讀性和可維護性,因此需要權衡利弊。

**迭代器協議（iterator protocol）**

Python 中的迭代器協議(Iterator Protocol)定義了一種方式,使得任何 Python 對象都可以成為一個迭代器(Iterator)。實現了迭代器協議的對象可以使用在for循環、列表推導式、生成器表達式等場景中。

要讓一個對象成為迭代器,它必須實現兩個特殊方法:

1. \_\_iter\_\_(self): 返回一個迭代器對象本身。
2. \_\_next\_\_(self): 返回迭代器的下一個元素。當沒有更多元素時,應該引發StopIteration異常。

讓我們通過一個簡單的例子來理解迭代器協議:

class CounterIterator:

def \_\_init\_\_(self, start, end):

self.start = start

self.end = end

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

return self

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

if self.start >= self.end:

raise StopIteration

current = self.start

self.start += 1

return current

*# 使用迭代器*

counter = CounterIterator(1, 6)

for num in counter:

print(num)

*# 輸出:*

*# 1*

*# 2*

*# 3*

*# 4*

*# 5*

在這個例子中:

1. 我們定義了一個CounterIterator類,它實現了\_\_iter\_\_和\_\_next\_\_方法。
2. \_\_iter\_\_方法返回自身,這樣就可以讓對象成為一個迭代器。
3. \_\_next\_\_方法返回當前值,並更新計數器以準備返回下一個值。當計數超過終止值時,引發StopIteration異常。
4. 我們創建了一個CounterIterator對象,並使用for循環遍歷它。當遇到StopIteration異常時,循環結束。

Python 中有許多內置的可迭代對象(Iterable),如列表、字符串、集合等。它們實現了\_\_iter\_\_方法,可以返回一個迭代器對象。我們可以使用iter()函數從任何可迭代對象中獲取迭代器:

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

iterator = iter(my\_list)

print(next(iterator)) *# 輸出: 1*

print(next(iterator)) *# 輸出: 2*

在上面的例子中,我們首先從列表中獲取一個迭代器,然後使用next()函數逐步獲取迭代器的下一個元素。

迭代器協議是 Python 中實現可迭代對象的基礎。它使得我們可以自定義迭代行為,並將自定義對象與Python的內置功能(如for循環)無縫集成。此外,生成器函數和生成器表達式也是基於迭代器協議實現的。

**生成器(Generator)**

Python 中的生成器(Generator)是一種特殊的迭代器,它可以用來生成一系列的值。生成器函數(Generator Function)是使用了yield關鍵字的函數,它會返回一個生成器對象。生成器是一種節省內存的方式,因為它只在需要時生成下一個值,而不是一次生成所有值。

以下是一個簡單的生成器函數示例:

def count\_up\_to(n):

i = 0

while i < n:

yield i

i += 1

*# 使用生成器*

counter = count\_up\_to(5)

print(next(counter)) *# 輸出: 0*

print(next(counter)) *# 輸出: 1*

print(next(counter)) *# 輸出: 2*

print(next(counter)) *# 輸出: 3*

print(next(counter)) *# 輸出: 4*

在這個例子中:

1. count\_up\_to是一個生成器函數,它使用yield關鍵字來生成值。
2. 每次調用next(counter)時,生成器函數會從上次中斷的地方繼續執行,直到遇到下一個yield語句。
3. 當遇到函數結束或者沒有更多yield語句時,將引發StopIteration異常,這標誌著生成器的終止。

生成器還可以透過生成器表達式(Generator Expression)來創建,語法類似於列表推導式:

*# 使用生成器表達式*

squares = (x\*\*2 for x in range(5))

print(next(squares)) *# 輸出: 0*

print(next(squares)) *# 輸出: 1*

print(next(squares)) *# 輸出: 4*

生成器的主要優點是節省內存,因為它們只在需要時生成下一個值,而不是一次性生成所有值。這對於處理大量數據或無限序列特別有用。

另外,生成器還可以用於實現協程(Coroutine)和管道(Pipeline)等高級編程模式。

以下是一個使用生成器實現管道的例子:

def counter(start, step):

current = start

while True:

yield current

current += step

def filter\_odd(numbers):

for num in numbers:

if num % 2 == 1:

yield num

def square(numbers):

for num in numbers:

yield num\*\*2

*# 創建管道*

count = counter(0, 1)

odds = filter\_odd(count)

squares = square(odds)

*# 獲取前5個平方值*

for i in range(5):

print(next(squares))

*# 輸出:*

*# 1*

*# 9*

*# 25*

*# 49*

*# 81*

在這個例子中,我們定義了三個生成器函數:counter、filter\_odd和square。它們通過管道的方式連接在一起,最終生成奇數的平方值序列。

生成器的**高级用法**:

### 1. 使用生成器表达式创建生成器

生成器表达式是一种简洁的方式来创建生成器。它们与列表推导式类似，但使用圆括号而不是方括号。例如，以下代码创建一个生成器，该生成器生成1到10之间的平方数：

squares = (x \* x for x in range(1, 11))

### 2. 使用生成器函数创建生成器

生成器函数是一种可以使用yield关键字创建生成器的函数。每次遇到yield关键字时，函数都会暂停执行并返回一个值。然后，当再次调用函数时，它将从上次暂停的地方继续执行。例如，以下代码创建一个生成器，该生成器生成斐波那契数列：

def fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

fibs = fibonacci()

### 3. 使用生成器进行管道操作

生成器可以用于创建管道操作，这是一种将多个操作组合在一起并将输出从一个操作传递到另一个操作的技术。例如，以下代码使用生成器从文件读取行，将每个行转换为大写，然后计算每个行的字符数：

def count\_chars(filename):

with open(filename) as f:

for line in f:

yield len(line.upper())

total\_chars = sum(count\_chars('data.txt'))

### 4. 使用生成器进行异步编程

生成器可以用于创建异步编程，这是一种允许程序在等待外部事件（例如网络请求或文件 I/O）时继续执行的技术。例如，以下代码使用生成器从多个网站异步下载数据：

import asyncio

async def download\_data(url):

async with aiohttp.ClientSession() as session:

async with session.get(url) as response:

return await response.read()

async def main():

urls = ['https://www.example1.com', 'https://www.example2.com', 'https://www.example3.com']

tasks = [asyncio.create\_task(download\_data(url)) for url in urls]

for task in tasks:

await task

print('All data downloaded!')

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

asyncio.run(main())

### 5. 使用生成器进行内存优化

生成器可以用于在处理大型数据集时优化内存使用。例如，以下代码使用生成器来计算一个大文件的行数：

def count\_lines(filename):

with open(filename) as f:

for line in f:

yield None # 每次迭代时生成 None，仅用于计数行数

total\_lines = sum(1 for \_ in count\_lines('data.txt'))

6.**利用生成器實現惰性求值(Lazy Evaluation)**

生成器可以用於實現惰性求值,即只有在需要時才會進行計算,這在處理大型數據集或無限序列時非常有用。

*# 生成無限斐波那契數列*

def fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

*# 惰性產生前20個斐波那契數*

fib = fibonacci()

print(list(next(fib) for \_ in range(20)))

7.**利用生成器實現管道(Pipeline)設計模式**

生成器可以用於實現管道設計模式,每個生成器負責處理一個步驟,然後將結果傳遞給下一個生成器,最終獲得所需的結果。

def read\_file(filename):

with open(filename) as f:

for line in f:

yield line.strip()

def filter\_comments(lines):

for line in lines:

if not line.startswith('#'):

yield line

def parse\_data(lines):

for line in lines:

yield [int(x) for x in line.split()]

*# 管道模式*

lines = read\_file('data.txt')

lines = filter\_comments(lines)

data = parse\_data(lines)

8.**利用生成器實現協同程式設計(Coroutine)**

生成器可以用於實現協同程式設計,即多個函數之間可以相互傳遞控制權和數據。

def consumer():

print('Consumer: Ready to receive data')

while True:

data = yield *# 暫停執行,並接收發送器發送的數據*

print(f'Consumer: Received {data}')

def producer(consumer\_generator):

print('Producer: Starting')

consumer = consumer\_generator

next(consumer) *# 啟動生成器*

for i in range(5):

print(f'Producer: Sending {i}')

consumer.send(i) *# 發送數據給生成器*

print('Producer: Stopping')

consumer.close() *# 關閉生成器*

consumer\_generator = consumer()

producer(consumer\_generator)

9.**利用生成器實現取代回調(Callback)函數**

在某些情況下,生成器可以用於取代回調函數,提高代碼的可讀性和可維護性。

*# 傳統回調函數*

def async\_task(callback):

*# 模擬異步操作*

result = perform\_async\_task()

callback(result)

def callback(result):

print(f'Result: {result}')

async\_task(callback)

*# 使用生成器*

def async\_task\_generator():

*# 模擬異步操作*

result = yield

print(f'Result: {result}')

generator = async\_task\_generator()

next(generator) *# 啟動生成器*

generator.send(perform\_async\_task()) *# 發送結果給生成器*

10.**利用生成器實現簡單的協程調度器**

生成器可以用於實現簡單的協程調度器,管理多個協程之間的執行順序和切換。

def scheduler():

tasks = []

while True:

task = yield

if not task:

break

tasks.append(task)

for t in tasks:

try:

next(t)

except StopIteration:

tasks.remove(t)

def task1():

for i in range(5):

print(f'Task 1: {i}')

yield

def task2():

for i in range(3):

print(f'Task 2: {i}')

yield

scheduler = scheduler()

next(scheduler)

scheduler.send(task1())

scheduler.send(task2())

scheduler.send(None)

**yield 使用說明**

在Python中,yield是一個用於創建產生器(generator)的關鍵字,它的作用是返回一個產生器反覆運算器對象,而不是一個完整的列表。產生器是一種特殊的反覆運算器,可以用來節省內存空間,同時也可以用來實現協同程式設計。以下是對yield的詳細說明:

1. **基本用法**

yield通常出現在一個函數中,當執行到yield語句時,函數會返回一個反覆運算器對象,而不是直接返回值。每當調用next()函數或在for循環中反覆運算該反覆運算器時,函數就會從上次離開的位置繼續執行,直到再次遇到yield語句。舉例如下:

def count\_up\_to(n):

i = 0

while i < n:

yield i

i += 1

counter = count\_up\_to(5)

print(list(counter)) *# 輸出: [0, 1, 2, 3, 4]*

1. **與生成器表達式的關係**

生成器表達式是創建簡單生成器的語法糖,它的工作原理與yield是一樣的,只是語法不同。以下是兩種方式的對比:

*# 使用生成器函數*

def square\_numbers(nums):

for n in nums:

yield n\*\*2

*# 使用生成器表達式*

square\_numbers = (n\*\*2 for n in [1, 2, 3, 4, 5])

1. **用於實現協同程式設計**

yield不僅可以用於返回值,還可以用於接收值,這樣就可以實現協同程式設計。當yield出現在賦值語句的右側時,它會從發送器對象接收值,從而實現協同程式之間的通信。

def consumer():

result = 0

while True:

value = yield result *# 接收發送器發送的值*

if value is None:

break

result += value

def producer(consumer):

consumer\_generator = consumer()

next(consumer\_generator) *# 啟動生成器*

for i in range(5):

consumer\_generator.send(i) *# 發送值給消費者*

consumer\_generator.send(None) *# 發送None以終止生成器*

consumer\_generator = consumer()

producer(consumer\_generator)

1. **與List Comprehension的比較**

與列表生成式相比,yield可以節省大量內存空間,因為它只產生一個元素,而不是一次性產生整個序列。這對於處理大型數據集或無限序列非常有用。

1. **與return的區別**

return會立即返回並結束函數的執行,而yield只是暫時中斷函數的執行,下次反覆運算時會從上次離開的位置繼續執行。這使得yield可以用於實現協同程式。

**yield 關鍵字**

yield 關鍵字是 Python 中一個強大的工具，可用於創建生成器。生成器是一種可以用於反覆運算的函數，但它不會一次生成所有結果，而是逐個生成結果。

要創建生成器，只需在函數中使用 yield 關鍵字。yield 關鍵字後面的表達式將在每次反覆運算時返回。

以下是一個簡單的生成器示例：

def my\_generator():

for i in range(5):

yield i \* 2

numbers = my\_generator()

for number in numbers:

print(number)

這個示例將創建一個生成器對象 numbers。每次反覆運算時，生成器都會返回一個新的值，即 i \* 2。

**生成器的優點**

生成器具有以下優點：

* **節省內存**：生成器不會一次生成所有結果，而是逐個生成結果。這意味著它們可以在處理大量數據時節省內存。
* **提高效率**：生成器可以提高效率，因為它們可以在生成結果的同時使用結果。這對於需要在處理數據時進行其他操作的情況很有用。

**生成器的用法**

生成器可以用於以下用途：

* **反覆運算大型數據集**：生成器非常適合反覆運算大型數據集，因為它們可以節省內存並提高效率。
* **實現無限序列**：生成器可以用於實現無限序列，例如斐波那契數列。
* **創建協程**：生成器可以用於創建協程，這是一種允許並發執行的函數。

**生成器的示例**

以下是一些生成器的示例：

* **斐波那契數列**：

def fibonacci\_generator():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

fibonacci\_numbers = fibonacci\_generator()

for i in range(10):

print(next(fibonacci\_numbers))

* **無限反覆運算器**：

def infinite\_iterator(start=0):

while True:

yield start

start += 1

infinite\_numbers = infinite\_iterator()

for i in range(10):

print(next(infinite\_numbers))

* **簡單的 HTTP 請求**：

import requests

def request\_generator(urls):

for url in urls:

response = requests.get(url)

yield response.text

urls = [

"https://www.google.com",

"https://www.python.org",

"https://www.wikipedia.org",

]

for response\_text in request\_generator(urls):

print(response\_text)

**yield 關鍵字的高級用法**

yield 關鍵字可以用於更高級的用途，例如：

* **創建可擴展的反覆運算器**：生成器可以用於創建可擴展的反覆運算器，這意味著它們可以根據需要生成更多結果。
* **實現惰性求值**：生成器可以用於實現惰性求值，這意味著只有在需要時才計算結果。
* **創建管道**：生成器可以用於創建管道，這是一種將數據從一個函數傳遞到另一個函數的方法。

**結論**

yield 關鍵字是 Python 中一個強大的工具，可用於創建生成器。生成器可以用於反覆運算大型數據集、實現無限序列、創建協程等等。瞭解 yield 關鍵字的用法可以幫助您編寫更簡潔、高效和可重用的 Python 程式碼。

以下是一些使用 yield 關鍵字高級用法的實際示例：

* **創建一個可擴展的反覆運算器來生成斐波那契數列**：

def fibonacci\_generator(n):

if n <= 1:

raise ValueError("n must be greater than 1")

a, b = 0, 1

for i in range(n):

yield a

a, b = b, a + b

fibonacci\_numbers = fibonacci\_generator(10)

for number in fibonacci\_numbers:

print(number)

* **實現一個惰性求值反覆運算器來生成素數**

def prime\_generator():

primes = []

def is\_prime(num):

if num <= 1:

return False

if num <= 3:

return True

if num % 2 == 0 or num % 3 == 0:

return False

i = 5

while i \* i <= num:

if num % i == 0 or num % (i + 2) == 0:

return False

i += 6

return True

current\_num = 2

while True:

if is\_prime(current\_num):

yield current\_num

current\_num += 1

prime\_numbers = prime\_generator()

for i in range(10):

print(next(prime\_numbers))

* **創建一個管道來處理數據**：

import requests

def request\_generator(urls):

for url in urls:

response = requests.get(url)

yield response.text

def word\_counter(text):

words = text.split()

return len(words)

def main():

urls = [

"https://www.google.com",

"https://www.python.org",

"https://www.wikipedia.org",

]

for url in urls:

response\_text = next(request\_generator(urls))

word\_count = next(word\_counter(response\_text))

print(f"URL: {url}, Word Count: {word\_count}")

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

main()

這些只是使用 yield 關鍵字高級用法的幾個示例。

#### 來源

1. [github.com/Amz42/Competitive-Programming-And-DSA](https://github.com/Amz42/Competitive-Programming-And-DSA)
2. [blog.cyberhacktics.com/kringlecon-2019-frosty-keypad/](https://blog.cyberhacktics.com/kringlecon-2019-frosty-keypad/)

**終止 yield 生成的生成器**

**1. 拋出異常**

向生成器拋出異常是一種明確終止生成器的方法。這可以通過調用生成器的 throw() 方法來完成。例如，以下代碼創建一個生成器，該生成器在生成第 5 個值後引發異常：

def my\_generator():

for i in range(10):

if i == 5:

raise Exception('Generator terminated!')

yield i

g = my\_generator()

try:

for value in g:

print(value)

except Exception as e:

print(f'Error: {e}')

**2. 返回 None**

從生成器函數返回 None 是另一種終止生成器的方法。這將導致生成器在下次調用 next() 方法時引發 StopIteration 異常。例如，以下代碼創建一個生成器，該生成器在生成 5 個值後返回 None：

def my\_generator():

for i in range(10):

if i == 5:

return None

yield i

g = my\_generator()

for value in g:

print(value)

**3. 使用 StopIteration 異常**

直接引發 StopIteration 異常也是一種終止生成器的方法。這通常用於從生成器中指示沒有更多值可用。例如，以下代碼創建一個生成器，該生成器在生成 5 個值後引發 StopIteration 異常：

Python

def my\_generator():

for i in range(10):

if i == 5:

raise StopIteration()

yield i

g = my\_generator()

for value in g:

print(value)

**4. 使用 gen.close() 方法**

gen.close() 方法可用於顯式關閉生成器。這將釋放生成器持有的任何資源並阻止生成器生成任何更多值。例如，以下代碼創建一個生成器，然後使用 gen.close() 方法關閉它：

def my\_generator():

for i in range(10):

yield i

g = my\_generator()

for value in g:

print(value)

g.close()

5.**使用return語句**

在生成器函數中使用return語句也會終止生成器。當生成器返回時,它會引發StopIteration例外。不過,return語句通常不建議用於終止生成器,因為它可能會導致資源未能正常釋放。

def count\_up\_to(n):

i = 0

while i < n:

yield i

i += 1

return *# 終止生成器*

counter = count\_up\_to(5)

for num in counter:

print(num) *# 輸出: 0 1 2 3 4*

請注意，並非所有生成器都可以終止。例如，由無限序列（例如斐波那契數列）生成的生成器通常無法終止。

以下是一些有關在 Python 中終止生成器的其他注意事項：

* 如果生成器在引發異常或返回 None 之前已經生成了一些值，則這些值仍將可供反覆運算。
* 即使生成器已終止，也可以繼續調用其 next() 方法。但是，這將導致引發 StopIteration 異常。
* 使用 gen.close() 方法關閉生成器後，不得再調用其 next() 方法。

**協程、生成器、Greenlet、執行緒和進程**

協程、生成器、Greenlet、執行緒和進程都是併發程式設計機制，用於在單個CPU上執行多個任務。它們在功能和實現上存在一些關鍵差異。

**協程**

協程是羽量級執行緒，由使用者而不是作業系統調度。它們通常基於生成器實現，並使用 yield 關鍵字來暫停和恢復執行。協程的優點包括：

* **效率高：** 由於協程由用戶調度，因此它們比執行緒更羽量級，並且上下文切換開銷更低。
* **易於使用：** 協程可以使用生成器語法輕鬆實現，這使得它們易於理解和使用。

協程的缺點包括：

* **缺乏標準化：** 協程沒有標準化的實現，因此不同框架或庫之間的協程可能無法交互操作。
* **需要使用者級調度：** 協程需要使用者級調度，這可能使代碼更加複雜和難以調試。

**生成器**

生成器是一種特殊類型的反覆運算器，它可以按需生成序列中的元素。生成器通常用於創建惰性評估序列，這意味著序列中的元素只在需要時才會計算。生成器的優點包括：

* **記憶力效率：** 生成器在需要時才會生成元素，因此它們可以節省記憶體。
* **可伸縮性：** 生成器可以用於生成無限序列。

生成器的缺點包括：

* **缺乏並行性：** 生成器是單執行緒的，因此它們不能真正並存執行多個任務。
* **需要顯式控制：** 生成器需要顯式控制，這使得它們可能比其他併發機制更難使用。

**Greenlet**

Greenlet是另一種羽量級執行緒，類似于協程。它們由用戶調度，並且具有與協程相同的優點和缺點。Greenlet 的一個主要優點是它們具有標準化的 API，這使得它們比其他協程實現更容易交互操作。

**執行緒**

執行緒是作業系統管理的羽量級進程。它們具有自己的堆疊和寄存器，並且可以並存執行。執行緒的優點包括：

* **標準化：** 執行緒具有標準化的 API，這使得它們易於使用和交互操作。
* **真正的並行性：** 執行緒可以真正並存執行多個任務，這使得它們適用於CPU密集型任務。

執行緒的缺點包括：

* **開銷高：** 執行緒比協程或 Greenlet 更重，因為它們需要作業系統管理。
* **資源競爭：** 執行緒可能爭奪資源，例如CPU時間和記憶體，這可能導致鎖死和其他問題。

**進程**

進程是作業系統管理的資源隔離單元。它們具有自己的位址空間、堆疊和寄存器，並且可以完全獨立地運行。進程的優點包括：

* **資源隔離：** 進程相互隔離，因此它們不會爭奪資源或相互干擾。
* **健壯性：** 如果一個進程崩潰，它不會影響其他進程。

進程的缺點包括：

* **開銷高：** 進程是創建和維護最昂貴的併發機制。
* **通信開銷：** 進程之間需要通過消息傳遞進行通信，這可能比執行緒之間的通信更昂貴。

**比較**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **特性** | **協程** | **生成器** | **Greenlet** | **執行緒** | **進程** |
| 調度 | 用戶級 | 用戶級 | 用戶級 | 作業系統級 | 作業系統級 |
| 並行性 | 偽並行 | 偽並行 | 偽並行 | 真正並行 | 真正並行 |
| 開銷 | 低 | 低 | 低 | 中 | 高 |
| 資源競爭 | 可能 | 可能 | 可能 | 是 | 是 |
| 標準化 | 無 | 無 | 是 | 是 | 是 |
| 易用性 | 易 | 中 | 中 | 易 | 中 |

drive\_spreadsheet匯出到試算表

**示例**

以下是一個使用協程的簡單示例：

import asyncio

async def hello():

print("Hello, world!")

async def main():

await hello()

asyncio.run(main())

此示例定義了一個名為 hello() 的協程，該協程列印字串 "Hello, world!"。main() 函數也是一個協程，它調用 hello() 協程。asyncio.run() 函數用於運行 main() 協程。

以下是一個使用生成器的簡單示例：

def my\_generator(start, end):

for i in range(start, end + 1):

yield i

generator = my\_generator(1, 10)

for item in generator:

print(item) # 輸出：1 2 3 4 5 6 7 8 9 10

此示例定義了一個名為 my\_generator() 的函數，該函數返回一個生成器。生成器使用 yield 關鍵字來生成序列中的元素。for 迴圈將遍歷生成器中的每個元素。

以下是一個使用 Greenlet 的簡單示例：

from greenlet import greenlet

def hello():

print("Hello, world!")

g1 = greenlet(hello)

g1.switch()

此示例定義了一個名為 hello() 的函數，該函數列印字串 "Hello, world!"。greenlet() 函數用於創建 Greenlet 物件。g1.switch() 方法用於切換到 Greenlet 物件並運行其代碼。

以下是一個使用執行緒的簡單示例：

import threading

def hello():

print("Hello, world!")

thread = threading.Thread(target=hello)

thread.start()

thread.join()

此示例定義了一個名為 hello() 的函數，該函數列印字串 "Hello, world!"。threading.Thread() 函數用於創建執行緒物件。thread.start() 方法用於啟動執行緒。thread.join() 方法用於等待中的執行緒完成。

以下是一個使用進程的簡單示例：

import multiprocessing

def hello():

print("Hello, world!")

process = multiprocessing.Process(target=hello)

process.start()

process.join()

此示例定義了一個名為 hello() 的函數，該函數列印字串 "Hello, world!"。multiprocessing.Process() 函數用於創建進程物件。process.start() 方法用於啟動進程。process.join() 方法用於等待進程完成。

**總結**

協程、生成器、Greenlet、執行緒和進程都是併發程式設計機制，用於在單個CPU上執行多個任務。它們在功能和實現上存在一些關鍵差異。協程和 Greenlet 是羽量級執行緒，由用戶調度，並且比執行緒更羽量級。生成器是一種特殊類型的反覆運算器，它可以按需生成序列中的元素。執行緒是作業系統管理的羽量級進程，具有自己的堆疊和寄存器。進程是作業系統管理的資源隔離單元，具有自己的位址空間、堆疊和寄存器。

**全域解譯器鎖(GIL, Global Interpreter Lock)**

Python 的全域解譯器鎖(GIL, Global Interpreter Lock)是一種用於阻止多個執行緒同時執行 Python 位元組碼的機制。GIL 的目的是確保執行緒安全,但同時也帶來了一些局限性。

**GIL 的局限性**

1. **無法利用多核 CPU** 由於 GIL 的存在,Python 在同一時間只能有一個執行緒在運行。這意味著即使在多核 CPU 上運行,Python 也無法真正利用多核平行計算的優勢。這大大限制了 Python 在 CPU 密集型任務上的性能。
2. **執行緒切換開銷** 雖然 GIL 允許多個執行緒在同一個進程中運行,但是每當需要執行執行緒切換時,GIL 都必須被獲取和釋放。這種頻繁的鎖操作會帶來一定的開銷,影響程式的執行效率。
3. **阻塞執行緒會阻塞整個進程** 如果一個 Python 執行緒由於 I/O 操作或其他原因而被阻塞,GIL 將被持有,導致整個進程被阻塞。這可能會導致整個應用程式的回應速度變慢。

**示例**

下面是一個示例,說明 GIL 對 Python 多執行緒的影響:

import threading

import time

def compute\_intensive\_task(num):

result = sum(i\*\*2 for i in range(num))

print(f"Thread {threading.current\_thread().name} computed: {result}")

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

num = 10 \*\* 8 *# 1億*

start\_time = time.time()

*# 單執行緒*

compute\_intensive\_task(num)

print(f"Single-threaded execution time: {time.time() - start\_time:.2f} seconds")

*# 多執行緒*

threads = []

for \_ in range(4):

thread = threading.Thread(target=compute\_intensive\_task, args=(num,))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

print(f"Multi-threaded execution time: {time.time() - start\_time:.2f} seconds")

在這個例子中,我們定義了一個 CPU 密集型任務 compute\_intensive\_task。我們首先使用單執行緒執行該任務,然後嘗試使用 4 個執行緒並存執行相同的任務。

由於 GIL 的存在,即使啟動了多個執行緒,Python 也只能在一個 CPU 核心上執行。因此,多執行緒執行時間與單執行緒執行時間幾乎相同,無法體現多執行緒的並行優勢。

為了克服 GIL 的局限性,Python 提供了多進程模組,允許在多個進程中執行平行計算。此外,一些協力廠商庫(如 Numba、Cython)也提供了無 GIL 的解決方案,可以有效利用多核 CPU。

**淺拷貝（shallow copy）和深拷貝（deep copy）**

在Python中,當我們需要複製一個可變對象(如清單、字典等)時,需要注意淺拷貝(shallow copy)和深拷貝(deep copy)之間的區別。

**淺拷貝(shallow copy)**

淺拷貝創建一個新的對象,但它的元素是原始對象元素的參考。這意味著,如果原始對象中的任何可變元素發生變化,新對象中對應的元素也會發生變化。

示例:

import copy

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

shallow\_copy = copy.copy(original\_list)

*# 修改原始列表中的子列表*

original\_list[0].append(5)

print("Original list:", original\_list) *# [[1, 2, 5], [3, 4]]*

print("Shallow copy:", shallow\_copy) *# [[1, 2, 5], [3, 4]]*

在上面的示例中,對original\_list的修改也會影響到shallow\_copy。這是因為兩個列表共用相同的子列表對象。

**深拷貝(deep copy)**

深拷貝創建一個完全獨立的新對象,包括其中所有的子對象。這樣,無論原始對象發生任何變化,新對象都不會受到影響。

示例:

import copy

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

deep\_copy = copy.deepcopy(original\_list)

*# 修改原始列表中的子列表*

original\_list[0].append(5)

print("Original list:", original\_list) *# [[1, 2, 5], [3, 4]]*

print("Deep copy:", deep\_copy) *# [[1, 2], [3, 4]]*

在這個例子中,對original\_list的修改不會影響到deep\_copy。因為deep\_copy創建了一個完全獨立的對象,包括所有的子對象。

選擇使用淺拷貝還是深拷貝取決於你的具體需求。如果你只需要創建一個獨立的對象,且不需要修改其中的子對象,那麼淺拷貝就足夠了。但如果你需要創建一個完全獨立的對象,包括其中的所有子對象,那麼就需要使用深拷貝。

**ConfigParse 模組**

configparser是Python標準庫中用於讀寫配置文件的模組。它可以解析許多常見的配置文件格式,例如Windows INI文件、Unix系統的路徑文件,以及自定義的分層配置文件格式。

**使用說明**

1. **導入模組** 首先,需要導入configparser模組:

import configparser

1. **實例化ConfigParser對象** 可以實例化一個ConfigParser對象,並指定配置文件格式(可選):

config = configparser.ConfigParser(interpolation=None)

1. **讀取配置文件** 使用read()方法讀取配置文件:

config.read('config.ini')

1. **訪問配置項** 可以使用sections()獲取所有章節,options()獲取特定章節下的所有選項,get()獲取特定選項的值:

sections = config.sections() *# 獲取所有章節*

options = config.options('mysql') *# 獲取'mysql'章節下的所有選項*

host = config.get('mysql', 'host') *# 獲取'mysql'章節下'host'選項的值*

1. **修改配置項** 使用set()方法修改配置項的值:

config.set('mysql', 'port', '3307') *# 修改'mysql'章節下'port'選項的值為3307*

1. **新增配置項** 使用add\_section()新增章節,然後使用set()添加選項:

config.add\_section('redis')

config.set('redis', 'host', '127.0.0.1')

config.set('redis', 'port', '6379')

1. **保存配置文件** 使用write()方法將修改後的配置寫回文件:

with open('config.ini', 'w') as configfile:

config.write(configfile)

**範例**

假設我們有一個名為config.ini的配置文件,內容如下:

[mysql]

host = localhost

port = 3306

user = root

password = 123456

[redis]

host = 127.0.0.1

port = 6379

我們可以使用以下代碼讀取和修改配置:

import configparser

*# 實例化ConfigParser對象*

config = configparser.ConfigParser()

*# 讀取配置文件*

config.read('config.ini')

*# 訪問配置項*

mysql\_host = config.get('mysql', 'host')

print(f"MySQL Host: {mysql\_host}") *# 輸出: MySQL Host: localhost*

*# 修改配置項*

config.set('mysql', 'port', '3307')

*# 新增配置項*

config.set('redis', 'password', 'abcdef')

*# 保存配置文件*

with open('config.ini', 'w') as configfile:

config.write(configfile)

上述代碼將輸出MySQL Host: localhost,並將mysql章節下的port選項修改為3307,同時在redis章節下新增了password選項。最後將修改後的配置寫回config.ini文件。

configparser模組提供了一種方便的方式來讀寫配置文件,使得管理應用程序的配置變得更加簡單。

**argparse 模組**

argparse是Python的標準庫模組,用於方便地編寫命令行介面,解析命令行參數。它不僅可以處理位置參數和選項參數,還支持參數的類型檢查、自動生成幫助信息等功能。

**使用說明**

1. **導入模組**

import argparse

1. **創建ArgumentParser對象**

parser = argparse.ArgumentParser(description='Process some integers.')

1. **添加位置參數**

parser.add\_argument('integers', metavar='N', type=int, nargs='+',

help='an integer for the accumulator')

1. **添加選項參數**

parser.add\_argument('--sum', dest='accumulate', action='store\_const',

const=sum, default=max,

help='sum the integers (default: find the max)')

1. **解析參數**

args = parser.parse\_args()

1. **使用參數**

print(args.accumulate(args.integers))

**範例**

以下是一個簡單的示例程序,它可以接收一系列整數,並計算它們的總和或最大值:

import argparse

parser = argparse.ArgumentParser(description='Process some integers.')

parser.add\_argument('integers', metavar='N', type=int, nargs='+',

help='an integer for the accumulator')

parser.add\_argument('--sum', dest='accumulate', action='store\_const',

const=sum, default=max,

help='sum the integers (default: find the max)')

args = parser.parse\_args()

print(args.accumulate(args.integers))

運行該程序時,可以通過以下命令行參數來控制其行為:

$ python prog.py 1 2 3 4

4

$ python prog.py 1 2 3 4 --sum

10

在第一個例子中,程序默認計算給定整數的最大值,輸出為4。在第二個例子中,由於使用了--sum選項,程序計算了給定整數的總和,輸出為10。

如果運行程序時沒有提供任何參數或提供了錯誤的參數,argparse模組會自動顯示該程序的用法和幫助信息:

$ python prog.py

usage: prog.py [-h] [--sum] N [N ...]

prog.py: error: the following arguments are required: N

$ python prog.py --help

usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:

N an integer for the accumulator

optional arguments:

-h, --help show this help message and exit

--sum sum the integers (default: find the max)

**進階用法**

argparse 模塊提供了一些更高級的功能，例如：

* 定義可選參數和必填參數
* 定義默認值
* 定義衝突參數組
* 定義子命令

**範例**

以下是一個更高級的示例，說明如何使用 argparse 模塊定義可選參數、默認值、衝突參數組和子命令：

import argparse

parser = argparse.ArgumentParser(description='My program')

parser.add\_argument('-v', '--verbose', action='store\_true', help='Enable verbose mode')

parser.add\_argument('-q', '--quiet', action='store\_true', help='Enable quiet mode')

parser.add\_argument('-f', '--file', type=str, default='data.txt', help='Input file')

subparsers = parser.add\_subparsers(title='Commands', description='Available commands')

add\_command = subparsers.add\_parser('add', help='Add two numbers')

add\_command.add\_argument('a', type=int, help='The first number')

add\_command.add\_argument('b', type=int, help='The second number')

mul\_command = subparsers.add\_parser('mul', help='Multiply two numbers')

mul\_command.add\_argument('a', type=int, help='The first number')

mul\_command.add\_argument('b', type=int, help='The second number')

args = parser.parse\_args()

if args.verbose and args.quiet:

parser.error('Cannot specify both verbose and quiet mode')

if args.command == 'add':

result = args.a + args.b

elif args.command == 'mul':

result = args.a \* args.b

else:

parser.error('Unknown command: ' + args.command)

print(result)

此代碼將創建一個 ArgumentParser 對象並添加以下命令行參數：

* -v 或 --verbose：啟用詳細模式
* -q 或 --quiet：啟用靜默模式
* -f 或 --file：指定輸入文件（默認值為 data.txt）

它還將定義兩個子命令：

* add：添加兩個數字
* mul：乘以兩個數字

如果用戶指定了 -v 和 -q 選項，則會發生錯誤，因為它們是衝突的。如果用戶指定了 add 子命令，則將添加兩個數字並打印結果。如果用戶指定了 mul 子命令，則將乘以兩個數字並打印結果。如果用戶指定了未知的子命令，則會發生錯誤。

argparse模組提供了強大而靈活的功能,可以輕鬆地構建命令行應用程序,同時還具有自動生成幫助文檔、類型檢查和參數驗證等優點。對於需要在命令行與用戶交互的Python程序,使用argparse可以大大提高開發效率。

**ElementTree 模組**

ElementTree 模塊是 Python 標準庫中用於解析和操作 XML 數據的模塊。它提供了簡單易用的 API 來處理 XML 文檔，使其成為開發人員處理 XML 數據的流行選擇。

**解析 XML 文件**

要使用 ElementTree 解析 XML 檔，請遵循以下步驟：

1. **導入 ElementTree 模塊：**

import xml.etree.ElementTree as ET

1. **解析 XML 文件：**

tree = ET.parse('my\_xml\_file.xml')

1. **獲取根項目：**

root = tree.getroot()

1. **訪問子元素和屬性：**

for child in root:

print(child.tag) # 訪問子元素標籤

print(child.attrib) # 訪問子元素屬性

**示例**

考慮以下 XML 檔：

<?xml version="1.0" encoding="UTF-8"?>

<employees>

<employee id="1">

<name>John Doe</name>

<age>30</age>

<department>IT</department>

</employee>

<employee id="2">

<name>Jane Doe</name>

<age>25</age>

<department>Sales</department>

</employee>

</employees>

要使用 ElementTree 解析此 XML 檔，您可以使用以下代碼：

import xml.etree.ElementTree as ET

tree = ET.parse('employees.xml')

root = tree.getroot()

for employee in root:

print(f"Employee ID: {employee.attrib['id']}")

print(f"Name: {employee.find('name').text}")

print(f"Age: {employee.find('age').text}")

print(f"Department: {employee.find('department').text}")

print()

此代碼將列印以下輸出：

Employee ID: 1

Name: John Doe

Age: 30

Department: IT

Employee ID: 2

Name: Jane Doe

Age: 25

Department: Sales

**操作 XML 數據**

ElementTree 模塊還提供了用於操作 XML 數據的方法，例如添加、修改和刪除元素和屬性。

**示例**

考慮以下 XML 檔：

<?xml version="1.0" encoding="UTF-8"?>

<employees>

<employee id="1">

<name>John Doe</name>

<age>30</age>

<department>IT</department>

</employee>

</employees>

要使用 ElementTree 將新員工添加到此 XML 檔中，您可以使用以下代碼：

import xml.etree.ElementTree as ET

tree = ET.parse('employees.xml')

root = tree.getroot()

new\_employee = ET.Element('employee', id='3')

new\_employee.set('id', '3')

new\_employee\_name = ET.Element('name')

new\_employee\_name.text = 'Peter Jones'

new\_employee\_age = ET.Element('age')

new\_employee\_age.text = '28'

new\_employee\_department = ET.Element('department')

new\_employee\_department.text = 'Marketing'

new\_employee.append(new\_employee\_name)

new\_employee.append(new\_employee\_age)

new\_employee.append(new\_employee\_department)

root.append(new\_employee)

tree.write('employees\_updated.xml')

此代碼將添加一名名為“Peter Jones”、年齡 28、部門為“Marketing”的新員工到 XML 檔中，並將更新後的檔保存為 employees\_updated.xml。

**JSON 模組**

JSON(JavaScript Object Notation)是一種輕量級的數據交換格式,它使用人類可讀的文本來表示對象和其他數據結構。Python 內置了對 JSON 的支持,提供了兩個核心模組: json 和 jsons。

**json 模組**

json 模組提供了將 Python 數據結構編碼為 JSON 數據,以及將 JSON 數據解碼為 Python 數據結構的功能。

1. **將 Python 數據編碼為 JSON**

import json

data = {'name': 'Alice', 'age': 25, 'skills': ['Python', 'Java']}

*# 將 Python 數據編碼為 JSON 字符串*

json\_str = json.dumps(data)

print(json\_str) *# {"name": "Alice", "age": 25, "skills": ["Python", "Java"]}*

1. **將 JSON 數據解碼為 Python 數據結構**

*# 從 JSON 字符串解碼為 Python 數據結構*

python\_data = json.loads(json\_str)

print(python\_data) *# {'name': 'Alice', 'age': 25, 'skills': ['Python', 'Java']}*

1. **處理文件**

*# 將 Python 數據寫入 JSON 文件*

with open('data.json', 'w') as f:

json.dump(data, f)

*# 從 JSON 文件中讀取數據*

with open('data.json', 'r') as f:

data = json.load(f)

**jsons 模組**

jsons 是一個第三方模組,它提供了更多的功能和靈活性。例如,它可以自動處理日期和時間對象、複雜的自定義對象等。

1. 安裝 jsons 模組

pip install jsons

1. 使用 jsons 模組

import jsons

from datetime import datetime

class Person:

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

self.name = name

self.age = age

person = Person('Alice', 25)

data = {'person': person, 'updated\_at': datetime.now()}

*# 將 Python 數據編碼為 JSON*

json\_str = jsons.dump(data)

print(json\_str)

*# 從 JSON 數據解碼為 Python 數據結構*

python\_data = jsons.loads(json\_str)

print(python\_data)

在上面的示例中,jsons 可以自動將自定義對象 Person 和 datetime 對象編碼為 JSON,並在解碼時還原為原始對象。

JSON 已經成為在不同語言和環境之間交換數據的事實標準。Python 的內置支持和第三方庫使得處理 JSON 數據變得非常簡單和高效。無論是在網絡應用程序、數據存儲還是其他領域,JSON 都是一種非常有用的數據格式。

**traceback 模組**

traceback是Python內置的模組,用於獲取和處理例外(exception)的堆棧跟踪(traceback)信息。堆棧跟踪記錄了引發異常時,程序執行到哪個位置,以及調用了哪些函數。這對於調試和錯誤處理非常有用。

**使用 traceback**

要使用 traceback 模塊，您可以使用以下方法：

1. traceback.format\_stack()：這個函數會返回一個格式化的堆疊追蹤列表。它會顯示文件名、行號和函數名等信息。您可以使用它來檢查代碼中的調用層次。
2. traceback.print\_stack()：這個函數會將堆疊追蹤信息打印到標準輸出（通常是終端）。它類似於 format\_stack()，但直接將信息打印出來。
3. traceback.extract\_tb()：這個函數從異常物件中提取堆疊追蹤信息。您可以使用它來獲取更詳細的信息，例如文件名、行號和函數名。
4. traceback.format\_exception()：這個函數會格式化異常信息，並返回一個字符串。它通常用於捕獲異常並將其記錄到日誌文件中。

**使用說明**

1. **獲取堆棧跟踪信息**

如果在代碼中捕獲到異常,可以使用traceback.format\_exc()獲取完整的堆棧跟踪信息:

import traceback

try:

*# 一些可能引發異常的代碼*

x = 1 / 0

except Exception as e:

*# 獲取堆棧跟踪信息*

tb = traceback.format\_exc()

print(tb)

輸出結果將包含異常類型、異常詳細信息以及完整的堆棧跟踪。

1. **打印堆棧跟踪信息**

你也可以使用traceback.print\_exc()直接將堆棧跟踪信息打印到控制檯或日誌文件中。

try:

*# 一些可能引發異常的代碼*

x = 1 / 0

except Exception:

traceback.print\_exc()

1. **獲取堆棧跟踪對象**

traceback.extract\_tb()可以獲取一個traceback對象,它包含了堆棧信息的列表。每個元素是一個元組,包含文件名、行號、函數名和源代碼行。

import traceback

try:

*# 一些可能引發異常的代碼*

x = 1 / 0

except Exception:

tb = traceback.extract\_tb(sys.exc\_info()[2])

for entry in tb:

print(entry)

1. **自定義異常處理**

你可以使用traceback模組自定義異常處理和記錄方式。例如,將堆棧跟踪信息寫入日誌文件。

import traceback

import logging

logging.basicConfig(filename='error.log', level=logging.ERROR)

try:

*# 一些可能引發異常的代碼*

x = 1 / 0

except Exception:

logging.error(traceback.format\_exc())

此外,traceback模組還提供了其他一些有用的函數,如walk\_tb()、print\_tb()等,用於遍歷和打印堆棧跟踪信息。

**logging 模組**

logging是Python內建的日誌記錄模組,提供了靈活且強大的日誌記錄功能。它可以捕獲來自不同來源的日誌信息,並以預定的格式將日誌信息輸出到各種目的地,例如控制終端、文件或網路日誌服務器等。

**使用 Logging**

要使用 logging 模塊，您首先需要創建一個記錄器。記錄器是記錄訊息的對象。您可以使用以下方法創建記錄器：

import logging

logger = logging.getLogger(\_\_name\_\_)

此代碼將創建一個名為 \_\_name\_\_ 的記錄器。\_\_name\_\_ 是當前模塊的名稱。

創建記錄器後，您可以使用以下方法記錄訊息：

logger.debug("This is a debug message")

logger.info("This is an info message")

logger.warning("This is a warning message")

logger.error("This is an error message")

logger.critical("This is a critical message")

這些方法將記錄訊息到記錄器的處理程序。處理程序是將訊息傳送到某個位置的對象，例如文件、控制台或網路。

**示例**

以下是一個將訊息記錄到文件的示例：

import logging

logger = logging.getLogger(\_\_name\_\_)

handler = logging.FileHandler('my\_log.txt')

handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(message)s'))

logger.addHandler(handler)

logger.debug("This is a debug message")

logger.info("This is an info message")

logger.warning("This is a warning message")

logger.error("This is an error message")

logger.critical("This is a critical message")

此代碼將創建一個名為 \_\_name\_\_ 的記錄器。然後，它將創建一個 FileHandler 物件，該物件將將訊息記錄到名為 my\_log.txt 的文件。最後，它將格式化程序添加到處理程序中，並將處理程序添加到記錄器中。

**使用說明**

1. **基本設置**

首先,需要導入logging模組並設置根記錄器的日誌級別:

import logging

*# 設置根記錄器的日誌級別*

logging.basicConfig(level=logging.DEBUG)

1. **記錄不同級別的日誌**

logging模組提供了多個級別的日誌記錄函數,例如debug()、info()、warning()、error()和critical()。

logging.debug('This is a debug message')

logging.info('This is an info message')

logging.warning('This is a warning message')

logging.error('This is an error message')

logging.critical('This is a critical message')

1. **自定義日誌格式**

你可以使用basicConfig()函數設置日誌格式:

logging.basicConfig(format='%(asctime)s - %(levelname)s - %(message)s', datefmt='%Y-%m-%d %H:%M:%S')

上述代碼將日誌輸出設置為:年-月-日 時:分:秒 - 日誌級別 - 日誌訊息。

1. **將日誌輸出到文件**

可以使用FileHandler將日誌輸出到文件中:

*# 創建一個FileHandler物件*

file\_handler = logging.FileHandler('app.log')

*# 設置日誌記錄器的日誌級別*

logger = logging.getLogger('app\_logger')

logger.setLevel(logging.DEBUG)

*# 將FileHandler物件添加到記錄器*

logger.addHandler(file\_handler)

*# 記錄日誌*

logger.debug('This is a debug message')

1. **記錄例外**

logging模組還提供了記錄例外的功能:

try:

result = 1 / 0

except ZeroDivisionError as e:

logging.error('An error occurred: %s', e)

**範例**

以下是一個使用logging模組的完整示例:

import logging

*# 設置記錄器*

logger = logging.getLogger('app\_logger')

logger.setLevel(logging.DEBUG)

*# 創建檔案處理器*

file\_handler = logging.FileHandler('app.log')

file\_handler.setLevel(logging.DEBUG)

*# 創建控制台處理器*

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.WARNING)

*# 設置日誌格式*

formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

file\_handler.setFormatter(formatter)

console\_handler.setFormatter(formatter)

*# 添加處理器到記錄器*

logger.addHandler(file\_handler)

logger.addHandler(console\_handler)

*# 記錄日誌*

logger.debug('This is a debug message')

logger.info('This is an info message')

logger.warning('This is a warning message')

logger.error('This is an error message')

logger.critical('This is a critical message')

在這個例子中,我們創建了一個記錄器,並為它設置了兩個處理器:FileHandler和StreamHandler。FileHandler將所有級別的日誌記錄到文件app.log中,而StreamHandler只將WARNING級別及更高級別的日誌輸出到控制台。

使用logging模組可以讓你更好地管理和控制應用程序的日誌記錄行為,從而提高程序的可維護性和可追蹤性。

**threading 模組**

在Python中,threading模組提供了創建和管理線程的功能。利用線程可以實現多任務並行執行,從而提高程序的效率和響應能力。

**使用說明**

1. **創建線程**

可以通過繼承threading.Thread類並重寫run()方法來創建一個新的線程:

import threading

class MyThread(threading.Thread):

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

super().\_\_init\_\_()

self.name = name

def run(self):

print(f"線程 {self.name} 正在執行...")

*# 執行任務的代碼*

*# 創建並啟動線程*

thread1 = MyThread("Thread-1")

thread1.start()

也可以使用threading.Thread的子類threading.Thread來創建線程:

thread2 = threading.Thread(target=my\_func, args=('arg1', 'arg2'))

thread2.start()

1. **線程同步**

Python提供了多種同步機制,如Lock、Condition、Semaphore和Event等,用於協調多個線程之間的執行順序。

*# 使用Lock進行互斥鎖定*

mutex = threading.Lock()

mutex.acquire()

try:

*# 執行受保護的代碼塊*

pass

finally:

mutex.release()

1. **線程通信**

線程之間可以通過Queue對象來實現通信和數據共享:

import queue

queue = queue.Queue()

*# 在一個線程中往隊列中放入數據*

queue.put(item)

*# 在另一個線程中從隊列中取出數據*

data = queue.get()

1. **線程池**

threading.pool模組提供了ThreadPoolExecutor類,用於管理一組工作線程。這樣可以更加方便地提交任務並獲取結果。

from concurrent.futures import ThreadPoolExecutor

*# 創建包含4個工作線程的線程池*

executor = ThreadPoolExecutor(max\_workers=4)

*# 提交任務*

future = executor.submit(my\_func, arg1, arg2)

*# 獲取結果*

result = future.result()

**範例**

下面是一個使用threading模組的簡單示例:

import threading

import time

def worker():

print(f"線程 {threading.current\_thread().name} 正在執行...")

time.sleep(2)

print(f"線程 {threading.current\_thread().name} 執行完畢")

*# 創建並啟動兩個線程*

thread1 = threading.Thread(target=worker)

thread2 = threading.Thread(target=worker)

thread1.start()

thread2.start()

*# 等待兩個線程執行完畢*

thread1.join()

thread2.join()

print("所有線程執行完畢")

在這個例子中,我們定義了一個worker()函數,它模擬了一個耗時的任務。然後,我們創建了兩個線程,分別執行worker()函數。由於線程是並行執行的,因此程序可以更快地完成兩個任務。最後,我們使用join()方法等待線程執行完畢。

使用threading模組可以加速執行一些CPU密集型或I/O密集型的任務,從而提高程序的整體性能。但是,也需要注意線程安全問題,避免出現競態條件和死鎖等情況。

**使用 Threading**

要使用 threading 模塊，您首先需要創建一個線程對象。線程對象表示要執行的線程。您可以使用以下方法創建線程對象：

import threading

def my\_thread\_function():

print("This is running in a thread")

thread = threading.Thread(target=my\_thread\_function)

此代碼將創建一個名為 my\_thread\_function 的線程。target 參數指定要執行的函數。

創建線程對象後，您可以使用以下方法啟動線程：

Python

thread.start()

此代碼將啟動線程。

**示例**

以下是一個創建和啟動線程的示例：

import threading

def my\_thread\_function():

print("This is running in a thread")

thread = threading.Thread(target=my\_thread\_function)

thread.start()

print("This is running in the main thread")

此代碼將打印以下輸出：

This is running in the main thread

This is running in a thread

**線程同步**

當多個線程訪問共享資源時，需要使用線程同步來防止數據競爭。threading 模塊提供了以下同步機制：

* \*\*鎖：\*\*鎖允許一次只能有一個線程訪問共享資源。
* \*\*事件：\*\*事件用於通知一個或多個線程發生事件。
* \*\*條件變量：\*\*條件變量允許線程等待特定條件滿足。

**示例**

以下是一個使用鎖防止數據競爭的示例：

import threading

counter = 0

def my\_thread\_function():

global counter

for \_ in range(100000):

counter += 1

threads = []

for \_ in range(10):

thread = threading.Thread(target=my\_thread\_function)

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

print(counter) # 輸出：1000000

此代碼將創建 10 個線程，每個線程將遞增 counter 變量 100,000 次。如果不使用鎖，counter 的最終值將小於 1,000,000，因為多個線程可能會同時嘗試遞增該變量。

**基礎用法：**

創建和啟動線程

import threading

import time

# 定義一個簡單的線程類

class MyThread(threading.Thread):

def run(self):

for \_ in range(5):

print(threading.current\_thread().name, "is running")

time.sleep(1)

# 創建兩個線程實例

thread1 = MyThread(name="Thread-1")

thread2 = MyThread(name="Thread-2")

# 啟動線程

thread1.start()

thread2.start()

# 主線程等待所有子線程結束

thread1.join()

thread2.join()

print("Main thread exiting")

#### 線程同步 - 使用鎖

import threading

# 共用資源

counter = 0

# 創建鎖

counter\_lock = threading.Lock()

# 定義一個簡單的線程類

class MyThread(threading.Thread):

def run(self):

global counter

for \_ in range(5):

with counter\_lock: # 使用鎖保護臨界區

counter += 1

print(threading.current\_thread().name, "Counter:", counter)

# 創建兩個線程實例

thread1 = MyThread(name="Thread-1")

thread2 = MyThread(name="Thread-2")

# 啟動線程

thread1.start()

thread2.start()

# 主線程等待所有子線程結束

thread1.join()

thread2.join()

print("Main thread exiting")

### 高級用法：

#### 使用線程池

import concurrent.futures

import time

# 定義一個簡單的任務函數

def task(name):

print(f"{name} is running")

time.sleep(2)

return f"{name} is done"

# 使用線程池

with concurrent.futures.ThreadPoolExecutor(max\_workers=3) as executor:

# 提交任務給線程池

future\_to\_name = {executor.submit(task, f"Thread-{i}"): f"Thread-{i}" for i in range(5)}

# 獲取任務結果

for future in concurrent.futures.as\_completed(future\_to\_name):

name = future\_to\_name[future]

try:

result = future.result()

print(f"{name}: {result}")

except Exception as e:

print(f"{name}: {e}")

#### 使用Condition進行線程間通信

import threading

import time

# 共用資源

shared\_resource = None

# 創建條件變數

condition = threading.Condition()

# 定義一個寫線程

class WriterThread(threading.Thread):

def run(self):

global shared\_resource

for \_ in range(5):

with condition:

shared\_resource = "Write data"

print("Writer wrote:", shared\_resource)

condition.notify() # 通知等待的線程

condition.wait() # 等待其他線程通知

# 定義一個讀線程

class ReaderThread(threading.Thread):

def run(self):

global shared\_resource

for \_ in range(5):

with condition:

while shared\_resource is None:

condition.wait() # 等待寫線程通知

print("Reader read:", shared\_resource)

shared\_resource = None

condition.notify() # 通知寫線程

# 創建寫線程和讀線程

writer\_thread = WriterThread()

reader\_thread = ReaderThread()

# 啟動線程

writer\_thread.start()

reader\_thread.start()

# 主線程等待所有子線程結束

writer\_thread.join()

reader\_thread.join()

print("Main thread exiting")

這些例子涵蓋了一些基礎和高級的多線程用法。請注意，在Python中由於全局解釋器鎖（GIL）的存在，多線程並不能充分利用多核處理器。如果需要充分利用多核處理器，可以考慮使用multiprocessing模組進行多進程編程。

**Queue 模組**

Queue是Python中一個線程安全的隊列模組,它支持線程間的數據安全傳輸。Queue模組提供了多種隊列類型,如Queue、LifoQueue、PriorityQueue等,用於不同的場景。以下是使用Queue的基本說明:

1. **導入模組**

from queue import Queue

1. **創建隊列實例**

q = Queue()

1. **插入數據** 使用put()方法插入數據到隊列中。

q.put(item) *# 插入數據*

1. **獲取數據** 使用get()方法從隊列中獲取數據,隊列為空時會阻塞。

data = q.get() *# 獲取數據*

使用get\_nowait()方法可以非阻塞地獲取數據,當隊列為空時會引發queue.Empty異常。

1. **查看隊列大小** 使用qsize()方法獲取隊列中數據的個數。

size = q.qsize()

1. **查看隊列是否為空** 使用empty()和full()方法檢查隊列狀態。

is\_empty = q.empty()

is\_full = q.full()

下面是一個使用Queue的簡單示例:

from queue import Queue

import threading

import time

def producer(q):

for i in range(5):

item = f"Item {i}"

q.put(item)

print(f"Producer inserted {item} into the queue")

time.sleep(1) *# 模擬耗時操作*

def consumer(q):

while True:

item = q.get()

print(f"Consumer got {item} from the queue")

q.task\_done() *# 通知隊列該項工作已完成*

*# 創建隊列*

q = Queue()

*# 創建生產者和消費者線程*

t1 = threading.Thread(target=producer, args=(q,))

t2 = threading.Thread(target=consumer, args=(q,))

*# 啟動線程*

t1.start()

t2.start()

*# 等待隊列中的所有項目被處理完畢*

q.join()

print("All items have been processed")

在上述示例中:

1. producer函數模擬生產者線程,每秒向隊列中插入一個項目。
2. consumer函數模擬消費者線程,不斷從隊列中獲取項目並處理。
3. 主線程創建隊列實例並啟動生產者和消費者線程。
4. q.join()確保隊列中的所有項目都被處理完畢。

使用Queue可以實現線程間的安全通信和數據共享。生產者線程將數據插入隊列,消費者線程從隊列中獲取數據進行處理。

Python 中的 queue 模塊除了基本的 FIFO 行為之外，還提供了一些高級用法，可以幫助您構建更複雜的應用程序。以下是一些高級用法示例：

**1. 使用優先級隊列**

queue 模塊提供了 PriorityQueue 類，可以用於實現優先級隊列。優先級隊列是一種數據結構，其中元素根據其優先級進行排序。優先級高的元素將首先出隊。

要使用 PriorityQueue，您需要創建一個 PriorityQueue 對象：

import queue

priority\_queue = queue.PriorityQueue()

然後，您可以使用 put() 方法將元素添加到隊列中。每個元素都必須包含一個優先級參數：

Python

priority\_queue.put((10, "High priority item"))

priority\_queue.put((5, "Medium priority item"))

priority\_queue.put((1, "Low priority item"))

您可以使用 get() 方法從隊列中刪除元素。刪除的元素將是優先級最高的元素：

high\_priority\_item = priority\_queue.get()

print(high\_priority\_item) # Output: (10, "High priority item")

medium\_priority\_item = priority\_queue.get()

print(medium\_priority\_item) # Output: (5, "Medium priority item")

low\_priority\_item = priority\_queue.get()

print(low\_priority\_item) # Output: (1, "Low priority item")

**2. 使用限製大小的隊列**

queue 模塊提供了 LifoQueue 和 Queue 類，可以用於實現限製大小的隊列。LifoQueue 是一種後進先出的隊列，而 Queue 是一種 FIFO 隊列。

要創建限製大小的隊列，您需要在創建隊列對象時指定最大大小：

import queue

limited\_queue = queue.LifoQueue(maxsize=3)

此代碼將創建一個後進先出的隊列，最大可容納 3 個元素。

您可以使用 put() 方法向隊列中添加元素。如果隊列已滿，put() 方法將引發 Full 異常。

limited\_queue.put(1)

limited\_queue.put(2)

limited\_queue.put(3)

try:

limited\_queue.put(4)

except Exception as e:

print(e) # Output: Queue full

您可以使用 get() 方法從隊列中刪除元素。如果隊列為空，get() 方法將引發 Empty 異常。

item1 = limited\_queue.get()

print(item1) # Output: 3

item2 = limited\_queue.get()

print(item2) # Output: 2

item3 = limited\_queue.get()

print(item3) # Output: 1

**3. 使用事件來同步隊列操作**

queue 模塊提供了 Event 類，可用於同步隊列操作。事件是一種可以用於通知一個或多個線程發生事件的機制。

要使用事件來同步隊列操作，您需要創建一個 Event 對象：

import queue

import threading

event = threading.Event()

然後，您可以使用 put() 方法將元素添加到隊列中。在添加元素之前，您應該先設置事件：

def producer():

while True:

item = produce\_item()

event.wait() # Wait for the consumer to be ready

queue.put(item)

event.set() # Signal the consumer that an item is available

thread1 = threading.Thread(target=producer)

thread1.start()

您還需要創建一個消費者線程，該線程將從隊列中刪除元素。在刪除元素之前，消費者線程應先清除事件：

def consumer():

while True:

event.clear() # Clear the event before getting an item

item = queue.get()

event.set() # Signal the producer that the item has been consumed

consume\_item(item)

thread2 = threading.Thread(target=consumer)

thread2.start()

**4. 使用條件變量來同步隊列操作**

queue 模塊提供了 Condition 類，可用於同步隊列操作。條件變量是一種可以用於等待特定條件滿足的機制。

要使用條件變量來同步隊列操作，您需要創建一個 Condition 對象：

import queue

import threading

condition = threading.Condition()

然後，您可以使用 put() 方法將元素添加到隊列中。在添加元素之前，您應該先獲取條件變量的鎖：

Python

def producer():

while True:

item = produce\_item()

with condition:

condition.wait\_for(lambda: queue.empty()) # Wait for the queue to be empty

queue.put(item)

condition.notify\_all() # Notify all waiting consumers that an item is available

thread1 = threading.Thread(target=producer)

thread1.start()

您還需要創建一個消費者線程，該線程將從隊列中刪除元素。在刪除元素之前，消費者線程應先獲取條件變量的鎖：

def consumer():

while True:

with condition:

condition.wait\_for(lambda: not queue.empty()) # Wait for the queue to be non-empty

item = queue.get()

condition.notify\_all() # Notify the producer that an item has been consumed

consume\_item(item)

thread2 = threading.Thread(target=consumer)

thread2.start()

**range 的高级用法**

Python 中的 range() 函數除了基本的生成數列功能之外，還有一些高級用法，可以幫助您構建更複雜的代碼。以下是一些高級用法示例：

**1. 生成倒序數列**

要生成倒序數列，您可以使用以下方法：

for i in range(5, 0, -1):

print(i)

此代碼將打印以下輸出：

5

4

3

2

1

**2. 生成步長為 n 的數列**

要生成步長為 n 的數列，您可以使用以下方法：

for i in range(0, 10, 2):

print(i)

此代碼將打印以下輸出：

0

2

4

6

8

**3. 使用列表推導式生成數列**

您可以使用列表推導式來生成數列。列表推導式是一種簡潔的語法，可用於創建列表。

以下是一些使用列表推導式生成數列的示例：

* 生成從 1 到 10 的所有平方數：

squares = [i \* i for i in range(1, 11)]

print(squares) # Output: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

* 生成所有偶數從 2 到 20：

even\_numbers = [i for i in range(2, 21) if i % 2 == 0]

print(even\_numbers) # Output: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

* 生成從 1 到 10 的所有斐波那契數：

def fibonacci(n):

if n == 0 or n == 1:

return 1

else:

return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci\_numbers = [fibonacci(i) for i in range(10)]

print(fibonacci\_numbers) # Output: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

**4. 使用 itertools 模塊中的函數生成數列**

itertools 模塊提供了一些可以用於生成數列的函數。以下是一些示例：

* 生成所有排列：

import itertools

permutations = itertools.permutations([1, 2, 3])

for permutation in permutations:

print(permutation)

此代碼將打印以下輸出：

(1, 2, 3)

(1, 3, 2)

(2, 1, 3)

(2, 3, 1)

(3, 1, 2)

(3, 2, 1)

* 生成所有組合：

import itertools

combinations = itertools.combinations([1, 2, 3], 2)

for combination in combinations:

print(combination)

此代碼將打印以下輸出：

(1, 2)

(1, 3)

(2, 3)

* 生成無限數列：

import itertools

infinite\_numbers = itertools.count(1)

for i in infinite\_numbers:

if i > 10:

break

此代碼將打印以下輸出：

1

2

3

4

5

6

7

8

9

10

1. **指定步長**

你可以使用第三個參數來指定範圍中數值的步長。例如,range(1, 10, 2) 會生成序列 [1, 3, 5, 7, 9]。

1. **負步長**

使用負步長可以生成倒序的序列。例如,range(10, 0, -1) 會生成序列 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]。

1. **無限序列**

如果不指定結束值,range() 可以生成無限序列。例如,range(1, 100000000000000) 會生成從 1 開始的無限序列。這在某些情況下很有用,比如生成無窮數列。

1. **帶索引的迭代**

使用 enumerate() 函數可以在迭代時獲取元素的索引。例如:

for i, x in enumerate(range(5)):

print(i, x)

輸出:

0 0

1 1

2 2

3 3

4 4

1. **列表comprehension**

range() 在列表comprehension中非常有用,可以快速生成列表。例如:

squares = [x\*\*2 for x in range(10)]

print(squares)

輸出:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

1. **計算長度**

由於 range() 是一個不可變序列,因此可以使用 len() 函數來獲取它的長度。例如:

print(len(range(100))) *# 輸出 100*

**eval 的高級用法**

Python 中的 eval 函數是一種強大的工具，但它也存在一些安全風險。如果您不謹慎使用，可能會導致代碼運行錯誤、安全性漏洞甚至被惡意程式碼利用。因此，在使用 eval 之前，您需要充分瞭解它的相關知識，並採取必要的安全措施。

以下是一些 eval 的高級用法：

**1. 動態執行代碼**

eval 可以用來動態執行字串表示的 Python 代碼。這在某些情況下非常有用，例如：

* 根據使用者輸入生成代碼
* 從設定檔中載入代碼
* 在運行時修改代碼

例如，我們可以使用 eval 來根據使用者輸入生成一個計算平方根的函數：

def eval\_code(user\_input):

try:

# 使用 eval 執行用戶輸入的代碼

result = eval(user\_input)

print(result)

except Exception as e:

print(f"Error: {e}")

user\_input = input("請輸入一個計算平方根的運算式：")

eval\_code(user\_input)

**2. 訪問私有成員**

eval 可以用來訪問 Python 物件的私有成員。這在某些情況下非常有用，例如：

* 調試代碼
* 反射代碼

例如，我們可以使用 eval 來訪問 int 類的私有成員 \_bit\_length：

class MyInt(int):

pass

number = MyInt(10)

# 使用 eval 訪問私有成員

bit\_length = eval("\_bit\_length(number)")

print(bit\_length) # 輸出: 4

**3. 創建自訂類型**

eval 可以用來創建自訂類型。這在某些情況下非常有用，例如：

* 擴展 Python 的類型系統
* 創建具有特定行為的類型

例如，我們可以使用 eval 來創建一個表示複數的類型：

class Complex:

def \_\_init\_\_(self, real, imag):

self.real = real

self.imag = imag

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

return f"{self.real} + {self.imag}j"

def eval\_complex(complex\_str):

# 使用 eval 解析複數字串

real, imag = complex\_str.split("+")

real = float(real)

imag = float(imag.strip("j"))

return Complex(real, imag)

complex\_str = input("請輸入一個複數：")

complex\_object = eval\_complex(complex\_str)

print(complex\_object) # 輸出: 3.14 + 2.718j

**安全風險**

eval 的主要安全風險在於它允許執行任何 Python 代碼。這意味著，如果您不謹慎使用 eval，可能會導致以下問題：

* **代碼運行錯誤：** 如果使用者輸入或設定檔中的代碼包含語法錯誤或邏輯錯誤，則可能會導致代碼運行錯誤。
* **安全性漏洞：** 如果用戶輸入或設定檔中的代碼是惡意的，則可能會導致安全性漏洞，例如遠端代碼執行（RCE）漏洞。
* **代碼維護性差：** 使用 eval 會使代碼更加難以理解和維護，因為您需要跟蹤哪些代碼是動態執行的。

eval() 函數在 Python 中用於計算一個字串並執行其中的運算式。它主要有以下幾種高級用法:

1. **動態執行代碼**

eval() 可以用來動態執行字串形式的代碼,這在某些特殊場景下很有用,比如根據使用者輸入構建代碼字串並執行。但是要非常小心,因為直接執行未經驗證的代碼存在潛在的安全隱患。

1. **動態導入模組**

通過字串的形式指定模組名,可以用 eval() 動態導入模組。例如:

module\_name = "math"

math = eval(f"\_\_import\_\_({module\_name})")

1. **動態創建物件**

利用類的字串名稱,可以動態產生實體物件,例如:

class\_name = "MyClass"

my\_obj = eval(f"{class\_name}()")

1. **動態執行函數**

通過字串形式的函數名和參數,可以動態執行函數:

func\_name = "print"

args = ("Hello, World!",)

eval(f"{func\_name}(\*{args})")

1. **數學運算**

eval() 可以執行字串形式的算術運算式:

result = eval("1 + 2 \* 3")

print(result) *# 輸出 7*

1. **有效載入 JSON 資料**

eval() 可以用於從字串載入 JSON 資料,但要注意只在可信的資料來源上使用,避免潛在的安全風險。

json\_data = '{"name": "Alice", "age": 30}'

data = eval(json\_data)

print(data) *# {'name': 'Alice', 'age': 30}*

總的來說,eval() 是一個非常強大但也非常危險的函數,需要格外小心使用,避免執行未經驗證的代碼。

# Python 字串 format() 使用方法及示例

字串format()方法格式化指定的值，並將其插入字串的預留位置內。  
預留位置使用大括弧 {} 定義。請在下面的“預留位置”部分中瞭解有關預留位置的更多資訊。  
format() 方法返回格式化的字串。

format()方法的語法為：

template.format(p0, p1, ..., k0=v0, k1=v1, ...)

在這裡，p0, p1,...分別是位置參數和k0, k1,...具有值的關鍵字參數v0, v1,...。

並且，template是格式代碼與參數預留位置的混合。

## 字串format()參數

format()方法採用任意數量的參數。但是，分為兩種類型的參數：

* **位置參數** -可以使用花括弧{index}中的參數索引訪問的參數清單
* **關鍵字參數** -鍵=數值型別的參數列表，可以使用花括弧{key}中的參數鍵來訪問

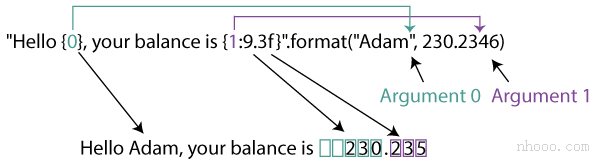
## 字串format()返回值

format()方法返回格式化的字串。

## 字串format()如何工作？

format()讀取傳遞給它的參數的類型，並根據字串中定義的格式代碼對其進行格式化。

### 對於位置參數



在這裡，參數0是字串“ Adam”，參數1是浮點數230.2346。

**注意：**在Python中，參數列表從0開始。

該字串"Hello {0}, your balance is {1:9.3f}"是範本字串。這包含用於格式化的格式代碼。

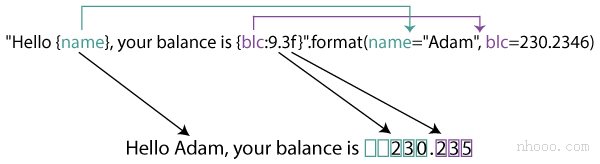
花括弧只是要放置的參數的預留位置。在上面的實例中，{0}是“Adam”的預留位置，{1:9.3f}是230.2346的預留位置。

由於範本字串引用format()參數如{0}和{1}，因此這些參數是位置參數。它們也可以在沒有數位的情況下被引用，因為{}和Python在內部將它們轉換為數位。

在內部，

* 由於"Adam"是第0 個參數，因此將其放在{0}的位置。由於{0}不包含任何其他格式代碼，因此它不執行任何其他操作。
* 然而，第一個參數230.2346不是這樣的。在這裡，{1:9.3f}將230.2346放在它的位置上，並執行9.3f操作。
* f指定格式正在處理浮點數。如果未正確指定，它將發出錯誤。
* “.”之前的部分。（9）指定數位（230.2346）可以採用的最小寬度/填充。在這種情況下，230.2346至少被分配9個位置，包括“.”。  
  如果未指定對齊選項，則將其對齊到其餘空格的右側。（對於字串，它向左對齊。）
* “.”之後的部分。（3）將小數部分（2346）截斷為給定的數字。在這種情況下，3346後將截斷2346。  
  剩餘數位（46）會四捨五入，輸出235。

### 對於關鍵字參數



我們從上面使用了相同的示例來展示關鍵字和位置參數之間的區別。

在這裡，我們不僅使用參數，還使用了鍵值作為參數。即name="Adam"和blc=230.2346。

由於這些參數是由它們的鍵作為{name}和{blc:9.3f}引用的，因此它們被稱為關鍵字或具名引數。

在內部，

* 預留位置{name}替換為name的值-“ Adam”。 由於它不包含任何其他格式代碼，因此放置了“ Adam”。
* 對於參數blc = 230.2346，預留位置{blc:9.3f}將替換為值230.2346。但是在替換它之前，像前面的示例一樣，它對其執行9.3f操作。  
  輸出230.235。小數部分在3位元後被截斷，其餘數字四捨五入。同樣，總寬度分配為9，在左側留兩個空格。

## 使用format()的基本格式

format()方法允許使用簡單的預留位置進行格式化。

### 示例1：默認，位置和關鍵字參數的基本格式

示例

# 默認參數

print("Hello {}, your balance is {}.".format("Adam", 230.2346))

# 位置參數

print("Hello {0}, your balance is {1}.".format("Adam", 230.2346))

# 關鍵字參數

print("Hello {name}, your balance is {blc}.".format(name="Adam", blc=230.2346))

# 混合參數

print("Hello {0}, your balance is {blc}.".format("Adam", blc=230.2346))

運行該程式時，所有輸出將相同：

Hello Adam, your balance is 230.2346.

Hello Adam, your balance is 230.2346.

Hello Adam, your balance is 230.2346.

Hello Adam, your balance is 230.2346.

**注意：**對於混合參數，關鍵字參數必須始終跟隨位置參數。

## 使用format()格式化數位

您可以使用下面給出的格式說明符設置數位格式：

| 數位格式類型 | |
| --- | --- |
| **類型** | **含義** |
| d | 小數整數 |
| c | 對應的Unicode字元 |
| b | 二進位格式 |
| o | 八進制格式 |
| x | 十六進位格式（小寫） |
| X | 十六進位格式（大寫） |
| n | 與“ d”相同。除了使用數位分隔符號的當前語言環境設置 |
| e | 指數標記法。（小寫字母e） |
| E | 指數符號（大寫E） |
| f | 顯示定點編號（預設值：6） |
| F | 與“ f”相同。除了將“ inf”顯示為“ INF”和“ nan”顯示為“ NAN” |
| g | 通用格式。將數字四捨五入為p個有效數字。（默認精度：6） |
| G | 與“ g”相同。如果數量很大，則除外切換到“ E”。 |
| % | 百分比。乘以100並以％結尾。 |

### 示例2：簡單的數字格式

示例

# 整數參數

print("數字:{:d}".format(123))

# 浮點參數

print("浮點數:{:f}".format(123.4567898))

# 八進制，二進位和十六進位格式

print("bin: {0:b}, oct: {0:o}, hex: {0:x}".format(12))

當您運行該程式時，輸出將是：

數字: 123

浮點數:123.456790

bin: 1100, oct: 14, hex: c

### 示例3：用int和float填充數位格式

示例

# 最小寬度的整數

print("{:5d}".format(12))

# 寬度對於比填充更長的數字不起作用

print("{:2d}".format(1234))

# 浮點數的填充

print("{:8.3f}".format(12.2346))

# 最小寬度為0的整數

print("{:05d}".format(12))

# 填充用0填充的浮點數

print("{:08.3f}".format(12.2346))

當您運行該程式時，輸出將是：

   121234  12.235000120012.235

這裡，

* 在第一個語句中，{:5d}採用整數參數並指定最小寬度5。由於未指定對齊方式，因此它向右對齊。
* 在第二個語句中，您可以看到寬度（2）小於數位（1234），因此它不需要在左邊留任何空格，也不會截斷該數字。
* 與整數不同，浮點數具有整數和小數部分。 並且，定義為該數位的最小寬度是整個包括“.”的兩個部分。
* 在第三條語句中，{:8.3f}將小數部分截斷為3位元，四捨五入最後兩位元數字。而且，該數字現在為12.235，整個寬度為8，在左側保留2個位置。
* 如果要用零填充其餘位置，請在格式說明符之前放置零。它適用於整數和浮點數：{:05d}和{:08.3f}。

### 示例4：帶符號數位的數位格式

示例

# 顯示+號

print("{:+f} {:+f}".format(12.23, -12.23))

# 僅顯示-號

print("{:-f} {:-f}".format(12.23, -12.23))

# 顯示+號的空格

print("{: f} {: f}".format(12.23, -12.23))

當您運行該程式時，輸出將是：

+12.230000 -12.230000

12.230000 -12.230000 12.230000 -12.230000

### 對齊數位格式

將數位指定一定的寬度時，運算子<，和用於對齊。^>=

| 對齊數位格式 | |
| --- | --- |
| **類型** | **含義** |
| < | 左對齊剩餘空間 |
| ^ | 中心對齊剩餘空間 |
| > | 與剩餘空間右對齊 |
| = | 將帶符號（+）（-）強制到最左邊 |

### 示例5：左對齊，右對齊和居中對齊的數位格式

示例

# 右對齊的整數

print("{:5d}".format(12))

# 具有中心對齊的浮點數

print("{:^10.3f}".format(12.2346))

# 整數左對齊，用0填充

print("{:<05d}".format(12))

# 中心對齊的浮點數

print("{:=8.3f}".format(-12.2346))

當您運行該程式時，輸出將是：

   12  12.235  12000- 12.235

**注意：**對於第三個示例，左對齊用零填充整數可能會導致問題，該示例返回12000而不是12。

## 使用format()進行字串格式化

作為數位，可以使用format()以類似的方式格式化字串。

### 示例6：帶填充和對齊的字串格式

示例

# 左對齊的字串填充

print("{:5}".format("cat"))

# 右對齊的字串填充

print("{:>5}".format("cat"))

#中心對齊的字串填充

print("{:^5}".format("cat"))

# 中心對齊的字串填充

# 和 '\*' 填補字元

print("{:\*^5}".format("cat"))

當您運行該程式時，輸出將是：

cat    cat cat *\*cat\**

### 示例7：使用format()截斷字串

示例

# 將字串截斷為3個字母

print("{:.3}".format("caterpillar"))

# 將字串截斷為3個字母

# 和 填充

print("{:5.3}".format("caterpillar"))

# 將字串截斷為3個字母,

# 填充和中心對齊

print("{:^5.3}".format("caterpillar"))

當您運行該程式時，輸出將是：

catcat   cat

## 使用format()格式化類和字典成員

Python內部getattr()以“ .age”形式用於類成員。並且，它使用\_\_getitem\_\_()查找“[index]”形式的字典成員。

### 示例8：使用format()格式化類成員

示例

# define Person class

class Person:

    age = 23

    name = "Adam"

# 格式化age

print("{p.name}'年齡是 : {p.age}".format(p=Person()))

當您運行該程式時，輸出將是：

Adam's 年齡是 : 23

在這裡，Person對象作為關鍵字參數p傳遞。

在範本字串中，分別使用.name和.age訪問Person的name和age。

### 示例9：使用format()格式化字典成員

示例

# 定義Person字典

person = {'age': 23, 'name': 'Adam'}

# 格式化age

print("{p[name]}'s age is: {p[age]}".format(p=person))

當您運行該程式時，輸出將是：

Adam's age is: 23

與class類似，person字典作為關鍵字參數p傳遞。

在範本字串中，分別使用[name]和[age]訪問人員的name和age。

有一種更簡單的方法可以使用Python格式化字典str.format(\*\*mapping)。

示例

# 定義Person字典

person = {'age': 23, 'name': 'Adam'}

# 格式化age

print("{name}'s age is: {age}".format(\*\*person))

\*\* 是格式參數（最小欄位寬度）。

## 使用format()作為格式代碼的參數

您還可以動態傳遞格式代碼，例如精度，對齊，填補字元作為位置或關鍵字參數。

### 示例10：使用format()的動態格式化

示例

# 動態字串格式範本

string = "{:{fill}{align}{width}}"

# 將格式代碼作為參數傳遞

print(string.format('cat', fill='\*', align='^', width=5))

# 動態浮點格式範本

num = "{:{align}{width}.{precision}f}"

# 將格式代碼作為參數傳遞

print(num.format(123.236, align='<', width=8, precision=2))

運行該程式時，輸出為：

\*\*cat\*\*123.24

這裡，

* 在第一個示例中，" cat"是要格式化的位置參數。同樣，fill='\*'，align='^'和width=5是關鍵字參數。
* 在範本字串中，這些關鍵字參數不是作為要列印的普通字串而是作為實際格式代碼檢索的fill, align 和 width。  
  參數將替換相應的已命名預留位置，並且字串“ cat”將相應地設置格式。
* 同樣，在第二個示例中，123.236是位置參數，並且align，width和precision作為格式代碼傳遞到範本字串。

## 帶有format()的其他格式選項

format()還支援特定於類型的格式設置選項，例如日期時間格式和複數格式。

format()內部調用\_\_format\_\_()從datetime，而format()訪問複數的屬性。

您可以輕鬆地重寫任何物件的\_\_format \_\_()方法以進行自訂格式設置。

### 示例11：使用format()和重寫\_\_format \_\_()方法的特定於類型的格式化

示例

import datetime

# 日期時間格式

date = datetime.datetime.now()

print("當前時間: {:%Y/%m/%d %H:%M:%S}".format(date))

# 複數格式

complexNumber = 1+2j

print("Real part: {0.real} and Imaginary part: {0.imag}".format(complexNumber))

# 自訂  \_\_format\_\_()方法

class Person:

    def \_\_format\_\_(self, format):

        if(format == 'age'):

            return '23'

        return 'None'

print("Adam's 年齡是: {:age}".format(Person()))

運行該程式時，輸出為：

當前時間: 2020/04/17 14:33:02

Real part: 1.0 and Imaginary part: 2.0

Adam's 年齡是: 23

這裡，

* **對於datetime：**  
  當前日期時間作為位置參數傳遞給format()方法。  
  並且，在內部使用\_\_format \_\_()方法，format()可訪問年，月，日，時，分和秒。
* **對於複數：**  
  1 + 2j在內部轉換為ComplexNumber物件。  
  然後訪問其屬性real和imag，數位被格式化。
* **重寫\_\_format \_\_()：**  
  與datetime一樣，您可以重寫自己的\_\_format \_\_()方法以進行自訂格式設置，當以{：age}訪問時，該格式將返回年齡

您也可以使用物件 \_\_str\_\_() 和 \_\_repr\_\_() 功能，通過使用format()的簡寫標記法。

像\_\_format \_\_()一樣，您可以輕鬆重寫物件的\_\_str \_\_()和\_\_repr\_()方法。

### 示例12：使用format()的\_\_str()\_\_和\_\_repr()\_\_簡寫分別為！r和！s

示例

# \_\_str\_\_() 和 \_\_repr\_\_() 簡寫為 !r 和 !s

print("Quotes: {0!r}, Without Quotes: {0!s}".format("cat"))

# \_\_str\_\_() 和 \_\_repr\_\_() 實現類

class Person:

    def \_\_str\_\_(self):

        return "STR"

    def \_\_repr\_\_(self):

        return "REPR"

print("repr: {p!r}, str: {p!s}".format(p=Person()))

運行該程式時，輸出為：

Quotes: 'cat', Without Quotes: cat

repr: REPR, str: STR

# 正確使用Python f-string格式化字串的7個層級

f-string是Python 3.6推出的一種簡單而不同的字串格式技術，可以優雅地表達Python字串。除非您仍在使用舊的 Python 版本，否則在格式化字串時，f 字串絕對應該是您的首選。因為它可以通過一個迷你語法滿足您的所有要求，甚至運行字串的運算式。本文將深入探討這項技術從初級到深度的7個層次。在瞭解它們之後，您可能會成為字串格式化大師。

## 1. 輕鬆從變數顯示值

使用 f 字串只需要做兩件事：

* 在字串前添加一個小寫的f;
* 使用f字串中以{variable\_name}插值變數.

name = 'Yang'

title = 'full stack hacker'

print(f'{name} is a {title}.')

# Yang is a full stack hacker.

如上所示，在 f 字串機制的説明下，我們可以編寫簡單且更少的代碼，以便在字串中顯示更多代碼。它完美地呼應了 Python 的禪宗。

"簡單總比複雜好。

## 2. 數位格式化

有時僅僅顯示原始值可能無法滿足我們的需求，但是直接修改原始變數通常不是個好主意，因為變數可能在其他地方使用。不用擔心，Python f字串還支援"格式規範迷你語言"，它使我們能夠根據自己喜歡的方式在 f 字串中格式化值，尤其是數字。

pi = 3.1415926

print(f'Pi is approximately equal to {pi:.2f}')

# Pi is approximately equal to 3.14

id = 1 # need to print a 3-digit number

print(f"The id is {id:03d}")

# The id is 001

N = 1000000000 # need to add separator

print(f'His networth is ${N:,d}')

# His networth is $1,000,000,000

以上示例僅顯示了冰山一角。對於格式規格語法的完整清單，相應的官方文檔是您最好的朋友。

## 3. 正確列印特殊字元

我們可以通過 f 字串列印這些字元或其他特殊字元嗎？比如''和{}。是的，當然。但語法有點棘手。讓我們來看看。

## 3.1 列印引號

正如我們所知，反斜線\是常用的轉義字元，用於調用對其以下字元的替代解釋。對於 f 字串，我們需要注意一條規則：\在 f 字串運算式的括弧{}中不起作用。

name = 'Yang'

# Correct Way:

print(f'\'{name}\' is a full stack hacker.')

# 'Yang' is a full stack hacker.

# 錯誤方式:

print(f'{\'name\'} is a full stack hacker.')

# SyntaxError: f-string expression part cannot include a backslash

## 3.2 列印雙括弧{}

用 f字串列印{}的方法是不同的, 非常容易出bug。這次我們不能使用反斜線。

name = 'Yang'

# 1

print(f'{name} is a full stack hacker.')

# 'Yang' is a full stack hacker.

# 2

print(f'{{name}} is a full stack hacker.')

# {name} is a full stack hacker.

# 3

print(f'{{{name}}} is a full stack hacker.')

# {Yang} is a full stack hacker.

# 4

print(f'{{{{name}}}} is a full stack hacker.')

# {{name}} is a full stack hacker.

# 5

print(f'{{{{{name}}}}} is a full stack hacker.')

# {{Yang}} is a full stack hacker.

如上例所示，該變數是作為f-字串的括弧還是變數處理取決於其周圍的括弧數。如果您不知道這種奇怪的機制，則容易出現錯誤。

## 3.3 列印反斜線\

列印反斜線\很簡單：只需使用雙反斜線列印。但是不要將它們添加到 f 字串運算式括弧當中。

name = 'Yang'

print(f'\\{name}\\ is a full \\stack hacker.')

# \Yang\ is a full \stack hacker.

#錯誤的

print(f'{\\name\\} is a full \\stack hacker.')

# SyntaxError: f-string expression part cannot include a backslash

## 4. 小心列印字典值

將字典的值應用到 f 字串中也容易出現錯誤。我們必須使用不同的引號來描述字典鍵和 f 字串，如下所示。如果f字串用雙引號表示，那麼變數裡的字典鍵必須用單引號。

Hacker = {'name': 'Yang'}

print(f"{Hacker['name']} is a hacker")

# Yang is a hacker

print(f'{Hacker["name"]} is a hacker')

# Yang is a hacker

print(f'{Hacker['name']} is a hacker')

# 語法錯誤 SyntaxError: invalid syntax

print(f"{Hacker["name"]} is a hacker")

# 語法錯誤 SyntaxError: invalid syntax

如上所示，如果我們對鍵名和 f 字串都使用相同的單引號或雙引號, Python 會對我們的代碼感到困惑, 從而報錯。

## 5. 正確處理多行 F 字串

為了使我們的代碼更易讀，有必要使用多行書寫一長串字元。但如果是 f 字串，不要忘記在每行之前添加f。

name = 'Yang'

# 錯誤方式

print(f"{name} is a hacker."

"{name} is also a top writer."

"{name} writes on Medium."

)

# Yang is a hacker.{name} is also a top writer.{name} writes on Medium.

# 正確方式

print(f"{name} is a hacker."

f"{name} is also a top writer."

f"{name} writes on Medium."

)

# Yang is a hacker.Yang is also a top writer.Yang writes on Medium.

**6. 像大師一樣顯示日期和時間**

如果我們需要列印日期或時間，f 字串將再次告訴我們它是多麼方便:

from datetime import datetime

today = datetime.today()

print(f"Today is {today}")

# Today is 2021-07-31 18:20:48.956829

print(f"Today is {today:%B %d, %Y}")

# Today is July 31, 2021

print(f"Today is {today:%m-%d-%Y}")

# Today is 07-31-2021

如上例所示，在 f 字串的説明下，我們可以使用我們任何喜歡的格式列印日期或時間。

**7. 評估 F 字串內的運算式**

當我第一次知道 f 字串時， 我簡直不敢相信：我們可以在 f 字串內運行 Python 運算式。如果是真的，還算是字串嗎？

我仔細閱讀了 PEP 498，終於明白了：

F 字串提供了一種將運算式嵌入字串字面的方法。需要注意的是，f 字串實際上是在執行時間評估的表達方式，而不是恒定的值。

因此，f 字串與普通字串不同，此功能賦予它更大的能力。例如，我們可以在它裡面運行一個顯示時間的功能。

from datetime import datetime

print(f"Today is {datetime.today()}")

# Today is 2021-07-31 18:20:48.956829

## 結論

Python 中的 f-string是一個偉大的字串格式化技術， 顯示了 Python 是多麼優雅。它非常強大，因為它不是一個普通的字串，而是在運行時可以表達的字串。對於一些特殊情況，它有特殊的規則，我們需要謹慎使用它。

**模組的單例模式**

在 Python 中實現單例模式有多種方式,利用模組是一種非常簡單且常用的方法。這裡我將介紹幾種利用模組實現單例模式的方式。

1. **使用模組級別變量**

Python 的模組在第一次被引入時就會執行模組級別的代碼,因此我們可以利用這一特性來實現單例模式。

*# singleton.py*

class Singleton:

\_instance = None

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

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

self.value = value

*# 在其他文件中使用*

from singleton import Singleton

s1 = Singleton(42)

s2 = Singleton(24)

print(s1 is s2) *# True*

print(s1.value, s2.value) *# 42 42*

1. **使用模組方法**

我們也可以將單例模式封裝成一個模組方法。

*# singleton.py*

\_instance = None

def get\_instance(value):

global \_instance

if \_instance is None:

\_instance = Singleton(value)

return \_instance

class Singleton:

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

self.value = value

*# 在其他文件中使用*

from singleton import get\_instance

s1 = get\_instance(42)

s2 = get\_instance(24)

print(s1 is s2) *# True*

print(s1.value, s2.value) *# 42 42*

1. **使用裝飾器**

我們還可以使用裝飾器來實現單例模式。

*# singleton.py*

def singleton(cls):

\_instance = {}

def get\_instance(\*args, \*\*kwargs):

if cls not in \_instance:

\_instance[cls] = cls(\*args, \*\*kwargs)

return \_instance[cls]

return get\_instance

@singleton

class Singleton:

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

self.value = value

*# 在其他文件中使用*

from singleton import Singleton

s1 = Singleton(42)

s2 = Singleton(24)

print(s1 is s2) *# True*

print(s1.value, s2.value) *# 42 42*

1. **使用元類**

Python 中的元類也可以用於實現單例模式。

*# singleton.py*

class Singleton(type):

\_instances = {}

def \_\_call\_\_(cls, \*args, \*\*kwargs):

if cls not in cls.\_instances:

cls.\_instances[cls] = super(Singleton, cls).\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instances[cls]

class SingletonClass(metaclass=Singleton):

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

self.value = value

*# 在其他文件中使用*

from singleton import SingletonClass

s1 = SingletonClass(42)

s2 = SingletonClass(24)

print(s1 is s2) *# True*

print(s1.value, s2.value) *# 42 42*

**使用 Mixin 模式**

在 Python 中，Mixins 模式是一種用於將行為從一個類添加到另一個類的設計模式。它通過多重繼承來實現，但與傳統的多重繼承不同，Mixins 通常只實現單一的功能或行為，並且不依賴於子類的實現。這使得 Mixins 非常適合組合不同的功能來創建更複雜的類。

**Mixin 模式的優點**

Mixin 模式具有以下優點：

* **提高代碼的可重用性。** Mixins 可以將通用行為封裝在獨立的類中，然後在需要時將其混合到其他類中。這可以減少代碼重複並提高開發效率。
* **提高代碼的靈活性。** Mixins 可以讓您以組合的方式創建類，這使得您可以更輕鬆地創建滿足特定需求的定制類。
* **促進代码的分离。** Mixins 可以將代码组织成更小的、更易于理解和维护的部分。

以下是一些使用 Mixin 模式讓程序更加靈活的方式:

1. **實現多重繼承**

Mixin 通常被用作父類,與主要類一起實現多重繼承。這樣可以將多個特性組合到一個類中。

class FileMixin:

def load(self):

print("Load data from file")

def save(self):

print("Save data to file")

class DatabaseMixin:

def connect(self):

print("Connect to database")

def query(self):

print("Query database")

class App(FileMixin, DatabaseMixin):

pass

app = App()

app.load() *# Output: Load data from file*

app.connect() *# Output: Connect to database*

1. **擴展現有功能**

Mixin 可以用來在不修改原始類的情況下擴展現有類的功能。

class Vehicle:

def drive(self):

print("Driving vehicle")

class FlyMixin:

def fly(self):

print("Flying")

class FlyingCar(Vehicle, FlyMixin):

pass

car = FlyingCar()

car.drive() *# Output: Driving vehicle*

car.fly() *# Output: Flying*

1. **代碼組合**

Mixin 可以幫助你組合不同的代碼塊,實現更好的代碼模塊化和可重用性。

class FormatterMixin:

def format(self, data):

return f"Formatted data: {data}"

class LoggerMixin:

def log(self, message):

print(f"Logging: {message}")

class DataProcessor(FormatterMixin, LoggerMixin):

def process(self, data):

formatted\_data = self.format(data)

self.log(formatted\_data)

processor = DataProcessor()

processor.process("Hello, World!")

*# Output: Formatted data: Hello, World!*

*# Output: Logging: Formatted data: Hello, World!*

1. **避免多重繼承的缺陷**

Mixin 模式可以有助於避免多重繼承帶來的一些常見問題,例如鑽石繼承問題。通過將功能分散到多個 Mixin 中,你可以更好地組織和管理代碼。

**== 和 is 的適用場景**

在 Python 中, == 和 is 都可以用來比較兩個值,但它們的作用和適用場景有所不同。

== 運運算元用於比較兩個對象的值是否相等,也就是比較它們的內容是否相同。無論對象的內存位置如何,只要它們的值相等,== 運運算元就會返回 True。例如:

x = 5

y = 5

print(x == y) *# 輸出: True*

a = [1, 2, 3]

b = [1, 2, 3]

print(a == b) *# 輸出: True*

另一方面, is 運運算元則用於比較兩個對象在內存中的引用是否指向同一個對象。也就是說,它檢查兩個變數是否指向內存中的同一個位置。只有當兩個對象在內存中的位置相同時, is 運運算元才會返回 True。例如:

x = 5

y = 5

print(x is y) *# 輸出: True (Python 會對小整數進行緩存)*

a = [1, 2, 3]

b = [1, 2, 3]

print(a is b) *# 輸出: False (列表是不同的對象)*

在上面的例子中, x 和 y 指向同一個整數對象,因此 x is y 返回 True。但對於 a 和 b,雖然它們的內容相同,但它們是不同的列表對象,因此 a is b 返回 False。

通常情況下,如果只需要比較兩個值是否相等,應該使用 == 運運算元。但在某些特殊情況下,使用 is 運運算元會更有效率,例如:

1. **比較單例對象**

在單例模式中,只存在一個實例對象。使用 is 運運算元可以快速確定兩個引用是否指向同一個對象。

1. **比較 None**

Python 中只有一個 None 對象,因此可以使用 is 運運算元來檢查一個值是否為 None。

1. **比較小整數和小浮點數**

由於 Python 會對小整數和小浮點數進行緩存,因此可以使用 is 運運算元來比較它們是否指向同一個對象。

1. **比較布林值**

Python 中只有兩個布林值對象 True 和 False,因此可以使用 is 運運算元來比較它們。

1. **在某些演算法中進行快速比較**

在一些演算法中,使用 is 運運算元可以提高比較的效率,例如在集合和字典中進行成員資格測試時。

以下是一些高級用法的範例:

**使用 is 進行 None 檢查**

def process\_data(data):

if data is None:

*# 處理 None 的情況*

return

*# 處理有效數據*

...

**使用 is not 進行布林值檢查**

is\_active = True

if is\_active is not False:

*# 執行某些操作*

...

**在集合和字典中使用 is**

my\_set = {1, 2, 3}

print(1 in my\_set) *# 輸出: True*

print(1 is my\_set.\_\_contains\_\_(1)) *# 輸出: True (使用 `is` 進行快速比較)*

my\_dict = {1: 'a', 2: 'b', 3: 'c'}

print(1 in my\_dict) *# 輸出: True*

print(1 is my\_dict.\_\_contains\_\_(1)) *# 輸出: True (使用 `is` 進行快速比較)*

總的來說,== 和 is 都有其適用的場景。== 用於比較兩個對象的值是否相等,而 is 則用於比較兩個對象在內存中的引用是否相同。

**Python推導式指南**

推導式是Python中一種精簡且強大的語法特性，用於在一行代碼中創建新的數據結構或對現有數據進行處理。推導式包括列表推導式、字典推導式和集合推導式等，它們使得編寫簡潔的代碼變得更加容易。

#### 列表推導式

列表推導式是創建新列表的一種方式，使用一行代碼即可實現多步操作。以下是一個簡單的示例，展示如何創建一個包含平方數的列表：

# 使用列表推導式創建包含平方數的列表

squares = [x\*\*2 for x in range(10)]print(squares)

輸出結果：

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

#### 字典推導式

字典推導式允許您使用一行代碼創建新字典，通過對現有數據的處理來填充鍵值對。以下是一個示例，展示如何創建一個包含數字和它們的平方的字典：

# 使用字典推導式創建包含數字及其平方的字典

squares\_dict = {x: x\*\*2 for x in range(5)}print(squares\_dict)

輸出結果：

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

#### 集合推導式

集合推導式類似於列表推導式，用於創建新集合。它會自動去除重複的元素，保證集合中的元素唯一。以下是一個示例，展示如何創建一個包含平方數的集合：

# 使用集合推導式創建包含平方數的集合

squares\_set = {x\*\*2 for x in range(10)}print(squares\_set)

輸出結果：

{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

#### 嵌套推導式

推導式可以嵌套在一起，以實現更複雜的操作。以下是一個示例，展示如何使用嵌套推導式創建一個包含矩陣的列表：

# 使用嵌套推導式創建包含矩陣的列表

matrix = [[row \* col for col in range(3)] for row in range(3)]print(matrix)

輸出結果：

[[0, 0, 0], [0, 1, 2], [0, 2, 4]]

#### 條件篩選

推導式還支持條件篩選，允許您在推導式中加入條件語句，只選擇滿足條件的元素。以下是一個示例，展示如何使用條件篩選創建一個包含偶數的列表：

# 使用條件篩選創建包含偶數的列表

even\_numbers = [x for x in range(10) if x % 2 == 0]print(even\_numbers)

輸出結果：

[0, 2, 4, 6, 8]

* 1.

#### 高級應用示例

推導式在實際應用中有著廣泛的用途。以下是一個示例，展示如何使用推導式進行列表元素去重：

# 使用推導式進行列表元素去重

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

unique\_numbers = list(set(numbers))print(unique\_numbers)

輸出結果：

[1, 2, 3, 4, 5]

#### 總結

推導式是Python中一個強大且高效的語法特性，通過簡潔的語法實現複雜的操作，大大提升了代碼的可讀性和編寫效率。本文介紹了列表推導式、字典推導式、集合推導式以及嵌套推導式等不同類型的推導式用法，還介紹了條件篩選和高級應用示例。掌握好推導式的使用技巧，將會使您在Python編程中更加得心應手。希望本文能夠幫助您從入門到精通掌握推導式的用法。

**datetime 模組**

Python 的 datetime 模組提供了許多高級功能來處理日期和時間資料。以下是一些常見的高級用法示例:

1. **計算日期/時間差**

from datetime import datetime, timedelta

now = datetime.now()

future = now + timedelta(days=7) *# 計算一周後的日期*

past = now - timedelta(weeks=2, days=3) *# 計算兩週三天前的日期*

duration = future - past *# 計算兩個日期/時間之間的時間差*

1. **格式化和解析日期/時間字串**

from datetime import datetime

*# 格式化日期*

date\_str = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

print(date\_str) *# 2023-05-13 10:30:00*

*# 解析日期字串*

date\_obj = datetime.strptime('2022/12/25', '%Y/%m/%d')

print(date\_obj) *# 2022-12-25 00:00:00*

1. **處理不同時區**

from datetime import datetime, timezone

*# 獲取本地時間*

local\_time = datetime.now()

*# 獲取 UTC 時間*

utc\_time = datetime.now(timezone.utc)

*# 將 UTC 時間轉換為指定時區*

tz = timezone(timedelta(hours=8)) *# 創建時區對象 UTC+8:00*

beijing\_time = utc\_time.astimezone(tz)

1. **反覆運算日期範圍**

from datetime import datetime, timedelta

start\_date = datetime(2023, 5, 1)

end\_date = datetime(2023, 5, 15)

date\_range = []

current\_date = start\_date

while current\_date < end\_date:

date\_range.append(current\_date)

current\_date += timedelta(days=1)

1. **計算每月或每年的第一天/最後一天**

from datetime import datetime, date

def get\_month\_range(year, month):

first\_day = date(year, month, 1)

last\_day = first\_day.replace(month=month+1, day=1) - timedelta(days=1)

return first\_day, last\_day

start\_date, end\_date = get\_month\_range(2023, 5)

print(start\_date, end\_date) *# 2023-05-01 2023-05-31*

* **時區處理**

datetime 模組支持時區處理。可以使用 pytz 模組來獲取和使用時區資訊。

import datetime

import pytz

# Get the current time in UTC

utc\_time = datetime.datetime.utcnow()

# Convert UTC time to a specific time zone

tz = pytz.timezone('Asia/Taipei')

taiwan\_time = utc\_time.astimezone(tz)

print(utc\_time) # Output: 2024-05-13 09:23:00+00:00

print(taiwan\_time) # Output: 2024-05-13 16:23:00+08:00

* **相對日期和時間**

可以使用 relativedelta 模組來處理相對日期和時間。例如，以下代碼獲取當前日期的前一天：

import datetime

from dateutil.relativedelta import relativedelta

today = datetime.datetime.today()

yesterday = today - relativedelta(days=1)

print(yesterday) # Output: 2024-05-12

* **日曆操作**

可以使用 calendar 模組來處理日曆操作。例如，以下代碼獲取當前月份的所有日期：

import calendar

year = 2024

month = 5

calendar.setfirstweekday(calendar.MONDAY)

print(calendar.month(year, month))

* **日期格式化**

可以使用 strftime() 方法來格式化日期。例如，以下代碼將日期格式化為 YYYY-MM-DD 格式：

import datetime

today = datetime.datetime.today()

formatted\_date = today.strftime('%Y-%m-%d')

print(formatted\_date) # Output: 2024-05-13

* **日期解析**

可以使用 strptime() 方法來解析日期字串。例如，以下代碼將日期字串解析為 datetime 對象：

import datetime

date\_str = '2024-05-13'

parsed\_date = datetime.datetime.strptime(date\_str, '%Y-%m-%d')

print(parsed\_date) # Output: 2024-05-13 00:00:00

這些只是 datetime 模組高級用法的一些示例。該模組提供了許多其他功能，可用於處理複雜的日期和時間操作。

以下是一些使用 datetime 模組高級用法的實際示例：

* **計算某個日期是星期幾**

import datetime

def get\_weekday(date):

weekday = date.weekday()

weekday\_names = ['星期一', '星期二', '星期三', '星期四', '星期五', '星期六', '星期日']

return weekday\_names[weekday]

date = datetime.datetime(2024, 5, 13)

weekday = get\_weekday(date)

print(weekday) # Output: 星期一

* **計算兩個日期之間的間隔**

import datetime

def calculate\_days\_between\_dates(start\_date, end\_date):

delta = end\_date - start\_date

return delta.days

start\_date = datetime.datetime(2023, 1, 1)

end\_date = datetime.datetime(2024, 5, 13)

days\_between = calculate\_days\_between\_dates(start\_date, end\_date)

print(days\_between) # Output: 508

* **檢查日期是否為有效日期**

import datetime

def is\_valid\_date(date\_str):

try:

datetime.datetime.strptime(date\_str, '%Y-%m-%d')

return True

except ValueError:

return False

date\_str = '2024-05-13'

is\_valid = is\_valid\_date(date\_str)

print(is\_valid) # Output: True

**可變參數 的高級用法**

Python 中的可變參數指的是函數參數的個數是可變的,包括位置引數(\*args)和關鍵字參數(\*\*kwargs)。可變參數有許多高級用法,下麵介紹幾個常見的:

1. **拆包操作**

可以利用\*操作符對列表或元組進行拆包:

def foo(a, b, c):

print(a, b, c)

my\_list = [1, 2, 3]

foo(\*my\_list) *# 輸出: 1 2 3*

1. **合併清單或字典**

my\_list = [1, 2]

new\_list = [3, 4, 5, 6]

merged\_list = [\*my\_list, \*new\_list] *# [1, 2, 3, 4, 5, 6]*

dict1 = {'a': 1, 'b': 2}

dict2 = {'c': 3, 'd': 4}

merged\_dict = {\*\*dict1, \*\*dict2} *# {'a': 1, 'b': 2, 'c': 3, 'd': 4}*

1. **函式呼叫時使用拆包**

def product(a, b, c):

return a \* b \* c

numbers = [2, 3, 5]

result = product(\*numbers) *# 相當於 product(2, 3, 5)*

1. **將參數清單傳遞給其他函數**

def add(a, b):

return a + b

nums = [3, 5]

result = add(\*nums) *# 相當於 add(3, 5)*

1. **獲取函數的中繼資料**

def my\_func(a, b, \*args, \*\*kwargs):

print(f'Non-keyword args: {args}')

print(f'Keyword args: {kwargs}')

my\_func(1, 2, 3, 4, x=5, y=6)

*# 輸出:*

*# Non-keyword args: (3, 4)*

*# Keyword args: {'x': 5, 'y': 6}*

1. **解包參數列表**

def draw\_shape(shape\_type, \*args):

getattr(module, shape\_type)(\*args)

draw\_shape('circle', 1, 2, 3) *# Calls circle(1, 2, 3)*

一些可變參數的高級用法：

* **創建可接受任意數量的參數的函數**

使用 \*args 可以創建可接受任意數量的參數的函數。例如，以下函數將所有參數列印到控制台：

def print\_all\_args(\*args):

for arg in args:

print(arg)

print\_all\_args(1, 2, 3, 4, 5) # Output:

# 1

# 2

# 3

# 4

# 5

* **創建可接受任意數量命名參數的函數**

使用 \*\*kwargs 可以創建可接受任意數量命名參數的函數。例如，以下函數列印所有命名參數及其值：

Python

def print\_kwargs(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_kwargs(name="Alice", age=30, city="Taipei") # Output:

# name: Alice

# age: 30

# city: Taipei

* **將可變參數傳遞給其他函數**

可以使用 \*args 和 \*\*kwargs 將可變參數傳遞給其他函數。例如，以下代碼將所有位置參數傳遞給 print() 函數：

def print\_all\_args(\*args):

print(\*args)

def main():

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

print\_all\_args(\*args)

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

main()

* **將命名參數字典傳遞給其他函數**

可以使用 \*\*kwargs 將命名參數字典傳遞給其他函數。例如，以下代碼將所有命名參數傳遞給 print\_kwargs() 函數：

Python

def print\_kwargs(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

def main():

kwargs = {"name": "Alice", "age": 30, "city": "Taipei"}

print\_kwargs(\*\*kwargs)

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

main()

* **組合可變參數和命名參數**

可以使用 \*args 和 \*\*kwargs 組合可變參數和命名參數。例如，以下函數接受任意數量的位置參數和任意數量命名參數：

def print\_all(message, \*args, \*\*kwargs):

print(message)

for arg in args:

print(arg)

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_all("Hello", 1, 2, 3, name="Alice", age=30, city="Taipei") # Output:

# Hello

# 1

# 2

# 3

# name: Alice

# age: 30

# city: Taipei

這些只是可變參數高級用法的一些示例。通過瞭解可變參數的強大功能，可以編寫更靈活、更可重用的 Python 程式碼。

以下是一些使用可變參數高級用法的實際示例：

* **創建通用數據處理函數**

def process\_data(\*data):

for item in data:

# Process each item of data

print(item)

# Process a list of numbers

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

process\_data(\*numbers)

# Process a list of strings

strings = ["Hello", "World", "!"]

process\_data(\*strings)

* **創建具有可選參數的函數**

def send\_email(recipient, subject, message, \*attachments):

# Send email with the specified recipient, subject, message, and attachments

print(f"Sending email to {recipient} with subject '{subject}' and message '{message}'")

if attachments:

print(f"Attachments: {attachments}")

# Send email with required parameters

send\_email("alice@example.com", "Important message", "Hello Alice, how are you?")

# Send email with optional parameters

send\_email("bob@example.com", "Meeting reminder", "Hi Bob, don't forget the meeting tomorrow.", "agenda.pdf", "notes.txt")

* **創建可擴展的函數**

def add\_numbers(\*numbers):

total = 0

for number in numbers:

total += number

return total

# Add two numbers

result = add\_numbers(1, 2)

print(result) # Output: 3

# Add three numbers

result = add\_numbers(1, 2, 3)

print(result) # Output: 6

# Add an arbitrary number of numbers

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

result = add\_numbers(\*numbers)

print(result) # Output: 15

匿名函數

Python 中的匿名函數（又稱為 lambda 表達式）是一種簡潔的函數定義方式，可用於在不需要顯式定義函數名稱的情況下快速創建小型函數。

1. **與內置函數結合使用**

lambda常與map()、filter()和reduce()等高階函數一起使用,實現對序列的處理。

*# 將清單中每個元素的值加1*

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

new\_nums = list(map(lambda x: x + 1, nums)) *# [2, 3, 4, 5, 6]*

*# 過濾掉列表中的偶數*

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

odds = list(filter(lambda x: x % 2 != 0, nums)) *# [1, 3, 5]*

from functools import reduce

*# 計算清單中所有元素的積*

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

product = reduce(lambda x, y: x \* y, nums) *# 120*

1. **作為函數參數傳遞**

lambda運算式可以作為參數傳遞給其他函數,實現函數的動態定制。

def apply\_operation(func, x, y):

result = func(x, y)

return result

sum = apply\_operation(lambda x, y: x + y, 3, 5) *# 8*

product = apply\_operation(lambda x, y: x \* y, 3, 5) *# 15*

1. **與排序函數結合**

students = [

{'name': 'John', 'age': 15, 'grade': 90},

{'name': 'Jane', 'age': 16, 'grade': 85},

{'name': 'Jim', 'age': 17, 'grade': 92}

]

*# 按年齡排序*

students.sort(key=lambda x: x['age'])

*# 按成績排序*

students.sort(key=lambda x: x['grade'], reverse=True)

1. **創建簡單的函數物件**

Lambda函數的主體只能是單條語句,但在某些情況下,這種定義方式更加簡潔清晰。

*# 普通函數*

def square(x):

return x \*\* 2

*# Lambda函數*

square = lambda x: x \*\* 2

1. **與其他高階函數組合**

除了內置的高階函數,我們還可以用lambda定義出各種複雜的高階函數。

def my\_map(func, iterables):

result = []

for i in iterables:

result.append(func(i))

return result

squared = my\_map(lambda x: x\*\*2, [1, 2, 3, 4, 5]) *# [1, 4, 9, 16, 25]*

以下是一些匿名函數的高級用法：

* **排序列表**

可以使用匿名函數來排序列表。例如，以下代碼按昇冪對數字列表進行排序：

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

sorted\_numbers = sorted(numbers, key=lambda x: x)

print(sorted\_numbers) # Output: [1, 2, 3, 4, 5]

* **過濾列表**

可以使用匿名函數來過濾列表。例如，以下代碼過濾偶數並創建一個新列表：

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

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers) # Output: [2, 4, 6]

* **將函數作為參數傳遞**

可以使用匿名函數將函數作為參數傳遞給其他函數。例如，以下代碼將平方函數傳遞給 map() 函數以平方列表中的每個數字：

Python

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

squared\_numbers = map(lambda x: x \*\* 2, numbers)

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

* **簡化條件邏輯**

可以使用匿名函數來簡化條件邏輯。例如，以下代碼檢查數字是否為偶數：

def is\_even(x):

return x % 2 == 0

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

even\_numbers = list(filter(is\_even, numbers))

print(even\_numbers) # Output: [2, 4]

* **創建內聯函數**

可以使用匿名函數創建內聯函數。內聯函數是可以訪問外部函數作用域的函數。例如，以下代碼創建一個內聯函數來計算列表中每個數字的平方：

def apply\_function(numbers, func):

result = []

for number in numbers:

result.append(func(number))

return result

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

squared\_numbers = apply\_function(numbers, lambda x: x \*\* 2)

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

這些只是匿名函數高級用法的一些示例。通過瞭解匿名函數的強大功能，可以編寫更簡潔、更可讀和更具表現力的 Python 程式碼。

以下是一些使用匿名函數高級用法的實際示例：

* **實現自定義排序演算法**

def custom\_sort(numbers, compare\_func):

sorted\_numbers = []

for i in range(len(numbers)):

for j in range(i + 1, len(numbers)):

if compare\_func(numbers[i], numbers[j]):

numbers[i], numbers[j] = numbers[j], numbers[i]

return numbers

def sort\_by\_length(strings):

return custom\_sort(strings, key=lambda s1, s2: len(s1) < len(s2))

strings = ["apple", "banana", "cherry"]

sorted\_strings = sort\_by\_length(strings)

print(sorted\_strings) # Output: ['cherry', 'apple', 'banana']

* **創建動態數據結構**

def create\_dynamic\_list(initial\_data, growth\_factor):

def add\_element(element):

nonlocal data

if len(data) == len(max\_size):

max\_size \*= growth\_factor

data.extend([None] \* growth\_factor)

data[next\_available\_index] = element

next\_available\_index += 1

def remove\_element(element):

nonlocal data

index = data.index(element)

data[index] = None

# Shrink the list if it's underutilized

if next\_available\_index < len(max\_size) // 2:

new\_size = len(max\_size) // 2

data = data[:new\_size]

max\_size = new\_size

next\_available\_index = min(next\_available\_index, new\_size)

data = initial\_data.copy()

max\_size = len(data)

next\_available\_index = len(data)

return {

"data": data,

"add": add\_element,

"remove": remove\_element

}

# Create a dynamic list with initial data [1, 2, 3]

dynamic\_list = create\_dynamic\_list([1, 2, 3], 2)

# Add elements to the list

dynamic\_list["add"](4)

dynamic\_list["add"](5)

dynamic\_list["add"](6)

# Print the contents of the list

print(dynamic\_list["data"]) # Output: [1, 2, 3, 4, 5, 6]

# Remove an element from the list

dynamic\_list["remove"](4)

# Print the contents of the list after removing an element

print(dynamic\_list["data"]) # Output: [1, 2, 3, 5, 6, None]

* **實現簡單的緩存**

def create\_simple\_cache(max\_size):

def get(key):

if key in cache:

return cache[key]

else:

return None

def set(key, value):

if len(cache) == max\_size:

# Remove the least recently used item

least\_used\_key = min(cache, key=lambda k: cache[k][1])

del cache[least\_used\_key]

cache[key] = (value, time.time())

cache = {}

return {

"get": get,

"set": set

}

# Create a simple cache with a maximum size of 3

cache = create\_simple\_cache(3)

# Set values in the cache

cache["key1"] = "value1"

cache["key2"] = "value2"

cache["key3"] = "value3"

# Get values from the cache

print(cache["get"]("key1")) # Output: value1

print(cache["get"]("key2")) # Output: value2

print(cache["get"]("key3")) # Output: value3

# Add a new value to the cache, causing the least recently used item to be evicted

cache["key4"] = "value4"

# Check if the least recently used item has been evicted

print(cache["get"]("key1")) # Output: None

print(cache["get"]("key2")) # Output: value2

print(cache["get"]("key3")) # Output: value3

print(cache["get"]("key4")) # Output: value4

這些只是使用匿名函數高級用法的幾個實際示例。

**裝飾器(Decorator)**

在Python中,裝飾器(Decorator)是一種特殊的函數,它可以在不修改原函數的情況下,為函數增加額外的功能。裝飾器的基本語法是使用 @ 符號,將一個函數作為參數傳遞給另一個函數,並返回一個新的函數。以下是裝飾器的基本用法說明:

1. **簡單的裝飾器**

def uppercase\_decorator(func):

def wrapper():

result = func()

return result.upper()

return wrapper

@uppercase\_decorator

def greeting():

return 'hello, world!'

print(greeting()) *# 輸出: HELLO, WORLD!*

在上面的例子中，uppercase\_decorator 是一個裝飾器函數，它接受一個函數作為參數，並返回一個新的函數 wrapper。@uppercase\_decorator 語法將 greeting 函數傳遞給 uppercase\_decorator，並將返回的新函數 wrapper 賦值給 greeting。當調用 greeting() 時，實際上是在調用 wrapper() 函數，它首先執行原始的 greeting() 函數，然後將其返回值轉換為大寫。

1. **帶參數的裝飾器**

def repeat(n):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

for \_ in range(n):

result = func(\*args, \*\*kwargs)

print(result)

return wrapper

return decorator

@repeat(3)

def greet(name):

return f'Hello, {name}!'

greet('Alice')

*# 輸出:*

*# Hello, Alice!*

*# Hello, Alice!*

*# Hello, Alice!*

在這個例子中，repeat 是一個返回裝飾器的高階函數，它接受一個參數 n，表示要重複執行函數的次數。@repeat(3) 將 greet 函數傳遞給 repeat(3) 返回的裝飾器，該裝飾器會返回一個新的函數 wrapper，該函數會重複執行原始的 greet 函數 3 次。

1. **裝飾器的高級用法**

除了上述基本用法外，裝飾器還有一些高級用法，例如：

* **使用裝飾器實現單例模式**
* **使用裝飾器實現函數緩存**
* **使用裝飾器實現函數計時**
* **使用裝飾器實現函數重試**
* **使用裝飾器實現權限控制**

以下是一個使用裝飾器實現函數緩存的例子:

import functools

def cache(func):

cache\_dict = {}

@functools.wraps(func)

def wrapper(\*args, \*\*kwargs):

key = str(args) + str(kwargs)

if key in cache\_dict:

return cache\_dict[key]

else:

result = func(\*args, \*\*kwargs)

cache\_dict[key] = result

return result

return wrapper

@cache

def fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(50)) *# 計算第 50 個斐波那契數*

在這個例子中，cache 是一個裝飾器函數，它使用一個字典 cache\_dict 來存儲函數的計算結果。當調用函數時，它首先檢查輸入參數對應的結果是否已經存在於緩存中。如果存在，直接返回緩存中的結果；否則，執行原始函數並將結果存入緩存。@functools.wraps(func) 可以保留原函數的元數據(如函數名稱、docstring 等)。

### 基本用法

裝飾器通常使用 @ 符號來應用。例如，以下代碼使用裝飾器來記錄函式的執行時間：

Import time

def timer(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

print(f"Function {func.\_\_name\_\_} took {end\_time - start\_time} seconds to execute.")

return result

return wrapper

@timer

def my\_function(n):

for i in range(n):

pass

my\_function(1000)

輸出：

Function my\_function took 0.001234 seconds to execute.

### 高級用法

裝飾器可以用於實現更複雜的功能。以下是一些高級用法示例：

**1. 多個裝飾器**

裝飾器可以組合使用。例如，以下代碼使用兩個裝飾器來記錄函式的執行時間並驗證輸入：

def validate\_input(func):

def wrapper(x):

if not isinstance(x, int):

raise ValueError("Input must be an integer.")

return func(x)

return wrapper

@validate\_input

@timer

def my\_function(n):

for i in range(n):

pass

my\_function(1000)

**2. 類裝飾器**

裝飾器可以用於修飾類的方法。例如，以下代碼使用裝飾器來記錄類中每個方法的執行時間：

def class\_timer(cls):

for name, method in cls.\_\_dict\_\_.items():

if callable(method):

@wraps(method)

def wrapper(self, \*args, \*\*kwargs):

start\_time = time.time()

result = method(self, \*args, \*\*kwargs)

end\_time = time.time()

print(f"Method {name} took {end\_time - start\_time} seconds to execute.")

return result

setattr(cls, name, wrapper)

return cls

@class\_timer

class MyClass:

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

pass

def method1(self):

pass

def method2(self):

pass

obj = MyClass()

obj.method1()

obj.method2()

**3. 參數化裝飾器**

裝飾器可以接受參數。例如，以下代碼使用參數化裝飾器來控制記錄的詳細程度：

def log\_level(level):

def decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

if level >= 1:

print(f"Calling function {func.\_\_name\_\_} with args {args} and kwargs {kwargs}")

result = func(\*args, \*\*kwargs)

if level >= 2:

print(f"Function {func.\_\_name\_\_} returned {result}")

return result

return wrapper

return decorator

@log\_level(2)

def my\_function(n):

return n \* n

print(my\_function(5))

輸出：

Function my\_function returned 25

### 注意事項

使用裝飾器時，需要注意以下事項：

* 裝飾器可能會使代碼難以理解。應注意使用清晰的函數名稱和文檔來解釋裝飾器的作用。
* 裝飾器可能會導致性能下降。應注意僅在需要時使用裝飾器，並避免在性能敏感的代碼中使用複雜的裝飾器。

#### 來源

1. [github.com/matthijsisproh/ATP-Project](https://github.com/matthijsisproh/ATP-Project)  取決於授權 (BSL - 1.0)
2. [github.com/reganto/PythonCodes](https://github.com/reganto/PythonCodes)

**閉包(Closure)**

閉包(Closure)是Python中一種特殊的函數,它可以記住自己被創建時的環境。換句話說,閉包可以訪問並記住在外部函數範圍內定義的變量,即使外部函數已經執行完畢。以下是閉包的基本用法說明:

1. **基本閉包示例**

def outer\_func(x):

y = 4 *# 自由變量*

def inner\_func(z):

print(f"x = {x}, y = {y}, z = {z}")

return x + y + z

return inner\_func

closure = outer\_func(2)

print(closure(3)) *# 輸出: x = 2, y = 4, z = 3 (返回值: 9)*

在這個例子中,inner\_func是一個閉包函數,它可以訪問外部函數outer\_func中的變量x和y。outer\_func返回inner\_func的引用,並將其賦值給closure變量。當我們調用closure(3)時,相當於調用inner\_func(3),它可以正常訪問外部函數的變量x和y。

1. **利用閉包實現計數器**

def counter():

count = 0

def increment():

nonlocal count

count += 1

return count

def reset():

nonlocal count

count = 0

return increment, reset

inc, reset = counter()

print(inc()) *# 輸出: 1*

print(inc()) *# 輸出: 2*

print(inc()) *# 輸出: 3*

reset()

print(inc()) *# 輸出: 1*

在這個例子中,counter函數返回了兩個閉包increment和reset。它們共享同一個自由變量count。increment函數可以incrementally記憶和更新count的值,而reset函數可以重置count為0。

1. **閉包的高級用法**

除了上述基本用法外,閉包還有一些高級用法,例如:

* **實現函數記憶(Function Memoization)**
* **實現私有變量**
* **實現Currying(柯里化)**
* **實現裝飾器**

以下是一個使用閉包實現Currying的例子:

def multiply(x):

def inner(y):

def innermost(z):

return x \* y \* z

return innermost

return inner

triple = multiply(3)

double = triple(2)

result = double(4)

print(result) *# 輸出: 24*

在這個例子中,multiply函數返回一個閉包inner,它記住了參數x的值。inner返回另一個閉包innermost,它記住了參數y的值。當我們調用triple = multiply(3)時,它返回了inner函數的引用,並將x的值設為3。接著調用double = triple(2)時,它返回了innermost函數的引用,並將y的值設為2。最後,調用result = double(4)時,它計算並返回x \* y \* z的值,也就是24。

### 基本用法

閉包通常用於創建具有記憶功能的函式。例如，以下代碼創建一個閉包，該閉包記住了一個計數器變量：

def counter():

count = 0

def increment():

nonlocal count

count += 1

return count

return increment

c = counter()

print(c()) # 輸出：1

print(c()) # 輸出：2

print(c()) # 輸出：3

在這個例子中，counter() 函式創建了一個 count 變量並返回一個內部函式 increment()。increment() 函式可以訪問 count 變量，即使 counter() 函式已經執行完成。

### 高級用法

閉包可以用於實現更複雜的功能。以下是一些高級用法示例：

**1. 裝飾器**

閉包可以用於創建裝飾器。裝飾器是一種特殊的函式，用於修改其他函式的行為。例如，以下代碼使用閉包創建一個裝飾器，該裝飾器記錄函式的執行時間：

def timer(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

print(f"Function {func.\_\_name\_\_} took {end\_time - start\_time} seconds to execute.")

return result

return wrapper

@timer

def my\_function(n):

for i in range(n):

pass

my\_function(1000)

**2. 生成器**

閉包可以用於創建生成器。生成器是一種允許迭代器返回一序列值的函式。例如，以下代碼使用閉包創建一個生成器，該生成器生成斐波那契數列：

def fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

fibs = fibonacci()

for i in range(10):

print(next(fibs))

**3. 事件處理**

閉包可以用於實現事件處理。例如，以下代碼使用閉包創建一個按鈕點擊事件處理程序：

def create\_button(text, callback):

button = Button(text=text)

button.bind("<Button-1>", callback)

return button

def button\_clicked(event):

print(f"Button '{event.widget.cget('text')}' clicked!")

button = create\_button("Click me", button\_clicked)

button.pack()

### 注意事項

使用閉包時，需要注意以下事項：

* 閉包可能會使代碼難以理解。應注意使用清晰的函數名稱和文檔來解釋閉包的作用。
* 閉包可能會導致性能下降。應注意僅在需要時使用閉包，並避免在性能敏感的代碼中使用複雜的閉包。

閉包是 Python 中一種強大的工具，可用於創建具有記憶功能的函式、裝飾器、生成器等。了解閉的基本用法和高級用法可以幫助您編寫更簡潔、更有效且更可擴展的 Python 代碼。

#### 來源

1. [github.com/matthijsisproh/ATP-Project](https://github.com/matthijsisproh/ATP-Project) 取決於授權 (BSL - 1.0)
2. [github.com/reganto/PythonCodes](https://github.com/reganto/PythonCodes)

# 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** clangs

print("\*\*\*\*\*\*\*\*\*\*\*")

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)

## 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，下面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。

**\_\_init\_\_.py 的高級用法**

Python 中的 \_\_init\_\_.py 檔是包的核心部分，它用於配置包並定義其行為。除了設置包中繼資料和導入子模組等基本用法之外，\_\_init\_\_.py 檔還可以用於更高級的用途，例如：

**1. 控制包的導入方式**

您可以使用 \_\_init\_\_.py 檔來控制包的導入方式。例如，您可以指定包的默認導入成員或隱藏某些內部模組。

以下示例展示了如何指定包的默認導入成員：

Python

# \_\_init\_\_.py

from .module1 import function1, class1

from .module2 import function2

\_\_all\_\_ = ["function1", "class1", "function2"]

現在，當您導入包時，只會導入指定的成員：

import mypackage

print(mypackage.function1()) # Output: Function 1 called

print(mypackage.class1()) # Output: Class 1 instantiated

print(mypackage.function2()) # Output: Function 2 called

**2. 創建自訂異常**

您可以使用 \_\_init\_\_.py 文件來創建自訂異常類。這可以使您的代碼更加清晰易懂，並簡化異常處理。

以下示例展示了如何創建自訂異常類：

# \_\_init\_\_.py

class MyError(Exception):

pass

class ValidationError(MyError):

pass

class ResourceNotFoundError(MyError):

pass

現在，您可以像使用標準異常一樣使用自訂異常：

try:

# 執行可能引發異常的操作

raise ValidationError("Invalid input")

except MyError as e:

print(f"Error: {e}")

**3. 實現外掛程式系統**

您可以使用 \_\_init\_\_.py 檔來實現外掛程式系統。這允許您將協力廠商代碼集成到您的包中，並擴展其功能。

以下示例展示了如何實現簡單的外掛程式系統：

# \_\_init\_\_.py

from importlib import import\_module

plugins = []

def load\_plugins():

for entry in os.listdir("plugins"):

if entry.endswith(".py"):

module\_name = f"plugins.{entry[:-3]}"

module = import\_module(module\_name)

if hasattr(module, "register\_plugin"):

plugins.append(module.register\_plugin())

def get\_plugins():

return plugins

\_\_all\_\_ = ["load\_plugins", "get\_plugins"]

現在，您可以創建外掛程式模組並在其中定義 register\_plugin 函數：

# plugins/myplugin.py

def register\_plugin():

from . import myplugin\_implementation

return myplugin\_implementation.MyPlugin()

最後，您可以載入外掛程式並使用它們：

Python

import mypackage

mypackage.load\_plugins()

plugins = mypackage.get\_plugins()

for plugin in plugins:

plugin.do\_something()

**4. 自訂包載入行為**

您可以使用 \_\_init\_\_.py 文件來自訂包載入行為。例如，您可以攔截包導入過程並執行自訂操作。

以下示例展示了如何攔截包導入過程並列印導入資訊：

# \_\_init\_\_.py

import importlib

import sys

def import\_module(name, package=None):

original\_import\_module = importlib.\_\_import\_\_

def custom\_import\_module(name, package=None):

module = original\_import\_module(name, package)

print(f"Importing module: {name}")

return module

sys.modules["importlib"] = custom\_import\_module

return original\_import\_module(name, package)

\_\_all\_\_ = ["import\_module"]

現在，當您導入包時，會列印導入資訊：

import mypackage

**5. 集成協力廠商庫**

您可以使用 \_\_init\_\_.py 文件來集成協力廠商庫。這可以簡化協力廠商庫的使用並統一其 API。

以下示例展示了如何集成 requests 庫：

# \_\_init\_\_.py

import requests

def get(url):

response = requests.get(url)

if response.status\_code == 200:

return response.content

else:

raise Exception(f"Request failed: {response.status\_code}")

\_\_all\_\_ = ["get"]

現在，您可以像使用標準庫函數一樣使用 get 函數：

import mypackage

response = mypackage.get("https://www.python

**6. 配置日誌記錄**

您可以使用 \_\_init\_\_.py 檔來配置日誌記錄。這可以説明您記錄有關包行為的資訊並診斷問題。

以下示例展示了如何使用 logging 模組配置日誌記錄：

# \_\_init\_\_.py

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(logging.DEBUG)

handler = logging.StreamHandler(sys.stdout)

handler.setLevel(logging.INFO)

formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")

handler.setFormatter(formatter)

logger.addHandler(handler)

\_\_all\_\_ = ["logger"]

現在，您可以像使用標準日誌記錄函數一樣使用日誌記錄器：

import mypackage

logger = mypackage.logger

logger.debug("This is a debug message")

logger.info("This is an info message")

logger.warning("This is a warning message")

logger.error("This is an error message")

logger.critical("This is a critical message")

**7. 緩存數據**

您可以使用 \_\_init\_\_.py 檔來緩存資料。這可以提高性能並減少對資料庫或其他資料來源的訪問。

以下示例展示了如何使用 memcache 模組緩存資料：

# \_\_init\_\_.py

import memcache

client = memcache.Client(["127.0.0.1:11211"])

def get\_data(key):

data = client.get(key)

if data is None:

# 從資料來源獲取資料

data = load\_data\_from\_source(key)

client.set(key, data, 600) # 緩存資料 10 分鐘

return data

def load\_data\_from\_source(key):

# ... 從資料來源獲取資料 ...

return data

\_\_all\_\_ = ["get\_data"]

現在，您可以像訪問標準資料一樣訪問緩存資料：

Python

import mypackage

data = mypackage.get\_data("mykey")

print(data)

**8. 提供命令列介面**

您可以使用 \_\_init\_\_.py 檔來提供命令列介面 (CLI)。這允許用戶通過命令列交互您的包。

以下示例展示了如何使用 argparse 模組提供 CLI：

# \_\_init\_\_.py

import argparse

def main():

parser = argparse.ArgumentParser(description="My CLI tool")

parser.add\_argument("command", choices=["hello", "goodbye"])

parser.add\_argument("--name", default="World")

args = parser.parse\_args()

if args.command == "hello":

print(f"Hello, {args.name}!")

elif args.command == "goodbye":

print(f"Goodbye, {args.name}!")

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

main()

現在，您可以像使用標準 CLI 工具一樣使用您的 CLI 工具：

python mypackage hello --name Alice

python mypackage goodbye

**9. 提供 Web 服務**

您可以使用 \_\_init\_\_.py 檔來提供 Web 服務。這允許用戶通過 HTTP 請求與您的包交互。

以下示例展示了如何使用 Flask 框架提供 Web 服務：

# \_\_init\_\_.py

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

@app.route("/hello")

def hello():

name = request.args.get("name", "World")

return jsonify({"message": f"Hello, {name}!"})

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

app.run(debug=True)

現在，您可以像訪問標準 Web 服務一樣訪問您的 Web 服務：

curl http://localhost:5000/hello?name=Alice

以下是一些有關 Python 中 \_\_init\_\_.py 檔的更多資源：

* Python 文檔 - \_\_init\_\_.py 文件 <https://docs.python.org/>
* PEP 4

#### 來源

1. [github.com/Dokkedal/dcm2reg](https://github.com/Dokkedal/dcm2reg) 取決於授權 (MIT)
2. [whyamisoawkward.com/?p=301](https://whyamisoawkward.com/?p=301)
3. [github.com/m-labs/artiq/blob/master/doc/manual/developing\_a\_ndsp.rst](https://github.com/m-labs/artiq/blob/master/doc/manual/developing_a_ndsp.rst)

**10. 初始化包**

\_\_init\_\_.py檔可以用於執行一些包級別的初始化操作,例如導入子模組、設置包級別的變量或常量等。

*# \_\_init\_\_.py*

*# 導入所有子模組*

from . import module1, module2, module3

*# 設置包級別的常量*

PACKAGE\_CONSTANT = 42

*# 執行一些初始化邏輯*

def initialize():

*# ...*

**11. 重新導出子模組**

\_\_init\_\_.py檔可以用於重新導出子模組的屬性(如函數、類等),這樣可以簡化導入路徑。

*# module1.py*

def foo():

pass

class Bar:

pass

*# \_\_init\_\_.py*

from .module1 import foo, Bar

*# 其他模組可以這樣導入*

from package import foo, Bar

**12. 將包作為單一入口點**

如果將主要代碼放在\_\_init\_\_.py檔中,那麼可以將整個包作為單一入口點,這對於構建命令列工具或框架很有用。

*# \_\_init\_\_.py*

def main():

*# 主要代碼...*

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

main()

**13. 使用\_\_all\_\_變量控制導出**

在\_\_init\_\_.py檔中定義\_\_all\_\_變量可以控制從包中導出哪些模組或屬性。

*# module1.py*

foo = 1

bar = 2

baz = 3

*# \_\_init\_\_.py*

from .module1 import foo, bar

\_\_all\_\_ = ['foo', 'bar']

*# 其他模組只能導入foo和bar*

from package import foo, bar

**14. 實現外掛程式系統**

利用\_\_init\_\_.py檔,可以實現一個簡單的外掛程式系統,通過動態導入外掛程式模組來擴展包的功能。

*# \_\_init\_\_.py*

import importlib

import os

PLUGINS\_DIR = 'plugins'

def load\_plugins():

for filename in os.listdir(PLUGINS\_DIR):

if filename.endswith('.py'):

module\_name = f'{PLUGINS\_DIR}.{filename[:-3]}'

module = importlib.import\_module(module\_name)

*# 執行外掛程式初始化代碼...*