# Python Cheat Sheet-9. (OOP)\_Classes\_Decorators(装饰器)\_Slots

?> PCS-9（    ）：9.1 装饰器-函数；9.2 装饰器-类；9.3 \_\_slots\_\_

知识点

描述

代码段

运算结果

备注

**9.1** 装饰器-函数

**1. 函数调用另一个函数（函数作为参数）**

这里定义了3个函数，say\_hello(name)和be\_awesome(name)，传入的为常规参数（不是以函数作为参数），并进行了不同方式字符串格式化。而对于greet\_bob(greeter\_func)，从greeter\_func("Bob")语句可以判断出函数参数greeter\_func为一个函数。将say\_hello(name)和be\_awesome(name)函数作为参数传入到函数greet\_bob(greeter\_func)，可以对应将参数替换为参数函数思考代码运行机制，比较方便理解。

该部分参考[Primer on Python Decorators](https://realpython.com/primer-on-python-decorators/)[1]

def say\_hello(name):  
 return f"Hello {name}" # f-string字符串格式化方法，Literal String Interpolation（文字字符串插值）  
  
def be\_awesome(name):  
 return f"Yo {name}, together we are the awesomest!"  
  
def greet\_bob(greeter\_func):  
 return greeter\_func("Bob")  
  
print(greet\_bob(say\_hello))  
print(greet\_bob(be\_awesome))

Hello Bob  
Yo Bob, together we are the awesomest!

**2.内置函数（inner functions）**

如果函数内部存在多个内置函数，函数定义的前后位置并不重要，主要由执行语句的顺序确定。同时，内部函数在调用父函数之前不会被定义，属于父函数parent()的局部作用域，仅在parent()内部作为局部变量存在。

def parent():  
 print("Printing from the parent() function")  
  
 def first\_child():  
 print("Printing from the first\_child() function")  
  
 def second\_child():  
 print("Printing from the second\_child() function")  
  
 second\_child()  
 first\_child()  
  
parent()

Printing from the parent() function  
Printing from the second\_child() function  
Printing from the first\_child() function

**3.函数返回值为一个函数**

python允许使用函数作为返回值，下述parent()函数返回了一个内置函数。需要注意，返回函数时，为return first\_child，是一个没有给()的函数名，意味返回first\_child函数的引用。如果给了()，则是返回first\_child函数的一个结果。当函数返回值为函数，则返回值（通常赋予于新的变量名）可以像普通函数一样调用（使用）。

def parent(num):  
 def first\_child():  
 return "Hi, I am Emma"  
  
 def second\_child():  
 return "Call me Liam"  
  
 if num == 1:  
 return first\_child  
 else:  
 return second\_child  
  
first=parent(1)  
second=parent(2)  
  
print(first)  
print(first())  
print(second())

<function parent.<locals>.first\_child at 0x000001B52F2933A0>  
Hi, I am Emma  
Call me Liam

**4.简单的装饰器**

say\_whee = my\_decorator(say\_whee)返回my\_decorator(func)父函数内置函数wrapper()的引用，为return wrapper。该内置函数包含父类传入的一个函数参数func，并执行func()，即执行参数函数的计算结果。执行say\_whee()时，即执行父类my\_decorator(func)内的wrapper()内置函数，只是此时，该函数已经独立于父函数my\_decorator(func)，并包含有执行wrapper()函数所需的所有参数，这里为参数函数func。因此，say\_whee = my\_decorator(say\_whee)中的say\_whee为一个闭包（Closure，或Lexical Closure），为一个结构体，存储了一个函数和与其关联的环境参数。

因为内置函数wrapper()，实际上对传入的参数函数say\_whee()的功能进行了增加，即“装饰”，所以可以简单说，装饰器就是对一个函数进行包装，修改已有的功能。

def my\_decorator(func):  
 def wrapper():  
 print("Something is happening before the function is called.")  
 func()  
 print("Something is happening after the function is called.")  
 return wrapper  
  
def say\_whee():  
 print("Whee!")  
  
say\_whee = my\_decorator(say\_whee)  
say\_whee()

Something is happening before the function is called.  
Whee!  
Something is happening after the function is called.

say\_whee = not\_during\_the\_night(say\_whee)装饰，则根据条件判断执行不同的操作，如果满足7 <= datetime.now().hour < 22，则执行外部函数say\_whee();否则，什么都不发生。

from datetime import datetime  
  
def not\_during\_the\_night(func):  
 def wrapper():  
 if 7 <= datetime.now().hour < 22:  
 func()  
 else:  
 pass # Hush, the neighbors are asleep  
 return wrapper  
  
def say\_whee():  
 print("Whee!")  
  
say\_whee = not\_during\_the\_night(say\_whee)  
say\_whee()

Whee!

**5.语法糖（Syntactic Sugar）**

上面的装饰器方法笨拙，为了简化代码过程，python允许用@symbol， 方式使用装饰器，有时称为pie语法。下述案例与上述结果一致，但是通过@my\_decorator pie方法，替代了say\_whee = not\_during\_the\_night(say\_whee)代码，简化操作。

def my\_decorator(func):  
 def wrapper():  
 print("Something is happening before the function is called.")  
 func()  
 print("Something is happening after the function is called.")  
 return wrapper  
  
@my\_decorator  
def say\_whee():  
 print("Whee!")  
  
say\_whee()

Something is happening before the function is called.  
Whee!  
Something is happening after the function is called.

**6.带参数的装饰器**

在wrapper\_do\_twice(\*args, \*\*kwargs)内置函数传入参数为\*args, \*\*kwargs， 接受任意数量的位置参数和关键字参数。并将其传入参数函数。

def do\_twice(func):  
 def wrapper\_do\_twice(\*args, \*\*kwargs):  
 func(\*args, \*\*kwargs)  
 func(\*args, \*\*kwargs)  
 return wrapper\_do\_twice  
  
  
@do\_twice  
def greet(name):  
 print(f"Hello {name}")  
   
greet("World")

Hello World  
Hello World

**7.装饰器的返回值**

如果装饰器要返回值，do\_twice(func)内置函数wrapper\_do\_twice(\*args, \*\*kwargs)在调用参数函数func时，需要执行return func(\*args, \*\*kwargs) 返回参数函数的返回值。

def do\_twice(func):  
 def wrapper\_do\_twice(\*args, \*\*kwargs):  
 func(\*args, \*\*kwargs)  
 return func(\*args, \*\*kwargs)  
 return wrapper\_do\_twice  
  
@do\_twice  
def return\_greeting(name):  
 print("Creating greeting")  
 return f"Hi {name}"  
  
hi\_adam = return\_greeting("Adam")  
print("-"\*50)  
print(hi\_adam)  
  
print("-"\*50)  
print(return\_greeting)  
print(return\_greeting.\_\_name\_\_)  
print("-"\*50)  
print(help(return\_greeting))

Creating greeting  
Creating greeting  
--------------------------------------------------  
Hi Adam  
--------------------------------------------------  
<function do\_twice.<locals>.wrapper\_do\_twice at 0x000001B52F293EE0>  
wrapper\_do\_twice  
--------------------------------------------------  
Help on function wrapper\_do\_twice in module \_\_main\_\_:  
  
wrapper\_do\_twice(\*args, \*\*kwargs)  
  
None

**8.保留原始函数的信息-自省（introspection）调整**

自省是指一个对象在运行时了解自己的属性的能力。例如，一个函数知道它自己的名字和文档。在上述示例中，通过return\_greeting.\_\_name\_\_，help(return\_greeting)等方式可以查看函数对象相关属性，但是，发现给出的是wrapper\_do\_twice的内置函数，而不是return\_greeting函数，因此可以通过[functools](https://docs.python.org/3/library/functools.html)① 的@functools.wraps(func)方法解决这个问题，保留原始函数的信息。

import functools  
  
def do\_twice(func):  
 @functools.wraps(func)  
 def wrapper\_do\_twice(\*args, \*\*kwargs):  
 func(\*args, \*\*kwargs)  
 return func(\*args, \*\*kwargs)  
 return wrapper\_do\_twice  
  
@do\_twice  
def return\_greeting(name):  
 print("Creating greeting")  
 return f"Hi {name}"  
  
hi\_adam = return\_greeting("Adam")  
print("-"\*50)  
print(hi\_adam)  
  
print("-"\*50)  
print(return\_greeting)  
print(return\_greeting.\_\_name\_\_)  
print("-"\*50)  
print(help(return\_greeting))

Creating greeting  
Creating greeting  
--------------------------------------------------  
Hi Adam  
--------------------------------------------------  
<function return\_greeting at 0x000001B530D988B0>  
return\_greeting  
--------------------------------------------------  
Help on function return\_greeting in module \_\_main\_\_:  
  
return\_greeting(name)  
  
None

**9. 带参数的装饰器**

装饰器中可以带参数，例如@repeat(num\_times=3)中num\_times=3。此时，对装饰器函数做了调整，增加了一层嵌套内置函数，传递装饰器参数。

def repeat(num\_times):  
 def decorator\_repeat(func):  
 @functools.wraps(func)  
 def wrapper\_repeat(\*args, \*\*kwargs):  
 for \_ in range(num\_times):  
 value = func(\*args, \*\*kwargs)  
 return value  
 return wrapper\_repeat  
 return decorator\_repeat  
  
@repeat(num\_times=3)  
def greet(name):  
 print(f"Hello {name}")  
   
greet("Galaxy")

Hello Galaxy  
Hello Galaxy  
Hello Galaxy

**10.多个装饰器装饰一个函数**

可以将多个装饰器堆叠在一起，应用在一个函数上，此时，执行的装饰器执行的顺序是从内到外，例如示例先执行@decor，返回值为20，而后再执行@decor1，返回值为400。

# code for testing decorator chaining  
def decor1(func):  
 def inner():  
 x = func()  
 return x \* x  
 return inner  
   
def decor(func):  
 def inner():  
 x = func()  
 return 2 \* x  
 return inner  
   
@decor1  
@decor  
def num():  
 return 10  
   
print(num())

400

**11.decorator模块简化装饰器**

使用[decorator模块](https://pypi.org/project/decorator/)②库的@decorator装饰器装饰‘装饰函数’，可以简化装饰器定义。例如下述代码取消了内置函数，将原始函数和输入参数都在do\_print(func,\*args, \*\*kwargs)，装饰函数中一起输入。

from decorator import decorator  
  
@decorator  
def do\_print(func,\*args, \*\*kwargs):  
 print('Hi {}!'.format(\*args,\*\*kwargs))  
 return func(\*args, \*\*kwargs)  
  
@do\_print  
def greet(name):  
 print(f"Hello {name}!")  
   
greet("World")

Hi World!  
Hello World!

**12.示例**

* 执行时间长度

这个装饰器存储函数开始运行前的时间start\_time = time.perf\_counter(), 和函数结束后的时间end\_time = time.perf\_counter()， 然后计算运行函数的时间，run\_time = end\_time - start\_time, 并打印。

import functools  
import time  
  
def timer(func):  
 """Print the runtime of the decorated function"""  
 @functools.wraps(func)  
 def wrapper\_timer(\*args, \*\*kwargs):  
 start\_time = time.perf\_counter() # 1  
 value = func(\*args, \*\*kwargs)  
 end\_time = time.perf\_counter() # 2  
 run\_time = end\_time - start\_time # 3  
 print(f"Finished {func.\_\_name\_\_!r} in {run\_time:.4f} secs")  
 return value  
 return wrapper\_timer  
  
@timer  
def waste\_some\_time(num\_times):  
 for \_ in range(num\_times):  
 sum([i\*\*2 for i in range(10000)])  
  
waste\_some\_time(999)

Finished 'waste\_some\_time' in 4.1239 secs

* 减缓运行

对执行的函数进行运行速度的限制。

import functools  
import time  
  
def slow\_down(func):  
 """Sleep 1 second before calling the function"""  
 @functools.wraps(func)  
 def wrapper\_slow\_down(\*args, \*\*kwargs):  
 time.sleep(1)  
 return func(\*args, \*\*kwargs)  
 return wrapper\_slow\_down  
  
@slow\_down  
def countdown(from\_number):  
 if from\_number < 1:  
 print("Liftoff!")  
 else:  
 print(from\_number)  
 countdown(from\_number - 1)  
   
countdown(3)

3  
2  
1  
Liftoff!

**9.2** 装饰器-类

**1.@property**

@property 内置装饰器可以将类的方法转换为只能读取的属性，例如使用andy.password类属性操作模式，而不是andy.password()类方法操作模式。如果要修改或者删除属性，则需要重新实现属性的setter，getter和deleter方法，例如@password.setter和@password.deleter装饰器。

class Bank\_acount:  
 def \_\_init\_\_(self):  
 self.\_password = 'preset password: 0000'  
  
 @property  
 def password(self):  
 return self.\_password  
  
 @password.setter  
 def password(self, value):  
 self.\_password = value  
  
 @password.deleter  
 def password(self):  
 del self.\_password  
 print('del complete')  
   
andy = Bank\_acount()  
print(andy.password) #getter  
andy.password='1q2w3e' #setter  
print(andy.password)  
del andy.password #deleter

preset password: 0000  
1q2w3e  
del complete

**2.@classmethod和@staticmethod**

类方法@classmetho和静态方法@staticmethod ，都可以直接通过Class/Instance.method()调用，可以不用实例化对象，直接由类直接调用，例如类方法的Person.fromBirthYear('mayank', 1996)和静态方法的Person.isAdult(22)。对于类方法，需要将self参数转换为cls；对于静态方法，则不需要self等任何参数。

示例迁移于[classmethod() in Python](https://www.geeksforgeeks.org/classmethod-in-python/)[2]

# Python program to demonstrate  
# use of a class method and static method.  
from datetime import date  
   
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
   
 # a class method to create a Person object by birth year.  
 @classmethod  
 def fromBirthYear(cls, name, year):  
 return cls(name, date.today().year - year)  
   
 # a static method to check if a Person is adult or not.  
 @staticmethod  
 def isAdult(age):  
 return age > 18  
   
person1 = Person('mayank', 21)  
person2 = Person.fromBirthYear('mayank', 1996)  
   
print(person1.age)  
print(person2.age)  
   
# print the result  
print(Person.isAdult(22))  
print(person1.isAdult(18))

21  
26  
True  
False

**3.@abstractmethod**

标准库[abc](https://docs.python.org/3/library/abc.html)③提供有@abstractmethod抽象方法，当所在的类继承了abc.ABC， 并给需要抽象的实例方法添加装饰器@abstractmethod后，这个类就成为了抽象类，不能够被直接实例化，例如示例的Animal类，抽象方法为info()。如果要使用抽象类，必须继承该类并实现该类的所有抽象方法，例如Bird子类继承了抽象类Animal，并在子类info()中实现父类抽象类的info()方法。

from abc import ABC, abstractmethod  
  
class Animal(ABC):  
 @abstractmethod  
 def info(self):  
 print("Animal")  
   
class Bird(Animal):  
 # 实现抽象方法  
 def info(self):  
 # 调用基类方法(即抽象方法)  
 super().info()  
 print("Bird")

animal = Animal()

---------------------------------------------------------------------------  
  
TypeError Traceback (most recent call last)  
  
Input In [65], in <cell line: 1>()  
----> 1 animal = Animal()  
  
  
TypeError: Can't instantiate abstract class Animal with abstract methods info

bird = Bird()  
bird.info()

Animal  
Bird

**4.装饰整个类**

装饰器接收的是一个类，而不是一个函数。

import functools  
import time  
  
def timer(func):  
 """Print the runtime of the decorated function"""  
 @functools.wraps(func)  
 def wrapper\_timer(\*args, \*\*kwargs):  
 start\_time = time.perf\_counter() # 1  
 value = func(\*args, \*\*kwargs)  
 end\_time = time.perf\_counter() # 2  
 run\_time = end\_time - start\_time # 3  
 print(f"Finished {func.\_\_name\_\_!r} in {run\_time:.4f} secs")  
 return value  
 return wrapper\_timer  
  
  
@timer  
class TimeWaster:  
 def \_\_init\_\_(self, max\_num):  
 self.max\_num = max\_num  
  
 def waste\_time(self, num\_times):  
 for \_ in range(num\_times):  
 sum([i\*\*2 for i in range(self.max\_num)])  
   
tw = TimeWaster(1000)  
tw.waste\_time(999)

Finished 'TimeWaster' in 0.0000 secs

**5.示例**

* 记录状态的装饰器

使用类作为装饰器，实现\_\_init\_\_()和\_\_call\_\_方法，完成函数运行状态的记录。

import functools  
  
class CountCalls:  
 def \_\_init\_\_(self, func):  
 functools.update\_wrapper(self, func)  
 self.func = func  
 self.num\_calls = 0  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 self.num\_calls += 1  
 print(f"Call {self.num\_calls} of {self.func.\_\_name\_\_!r}")  
 return self.func(\*args, \*\*kwargs)  
  
@CountCalls  
def say\_whee():  
 print("Whee!")  
   
say\_whee()  
say\_whee()  
say\_whee()

Call 1 of 'say\_whee'  
Whee!  
Call 2 of 'say\_whee'  
Whee!  
Call 3 of 'say\_whee'  
Whee!

**9.3** \_\_slots\_\_

通过\_\_slots\_\_类属性分配一连串的字符串属性名称进行属性声明，从而限制类实例对象将拥有的合法属性集，达到优化内存，提高程序运行速度的作用。当为\_\_slots\_\_分配一串字符串名称，则只有\_\_slots\_\_列表中的那些名称可以被分配为实例属性，并在实例化时，阻止了为实例分配\_\_dict\_\_对象，除非在\_\_slots\_\_中包含该对象。

下述案例类IceTeaSales中配置\_\_slots\_\_对象的属性名称包括['temperature','iceTeaSales']，因此当配置非该列表中所列的属性名，例如iceTea.price时，就会引发异常。

class IceTeaSales:  
 \_\_slots\_\_=['temperature','iceTeaSales']  
 def \_\_init\_\_(self):  
 self.temperature=0  
 self.iceTeaSales=0  
   
iceTea=IceTeaSales()  
print(iceTea.temperature)  
iceTea.temperature=29  
setattr(iceTea,'iceTeaSales',77)  
print(iceTea.iceTeaSales,iceTea.temperature)  
print(getattr(iceTea,'temperature'))  
iceTea.price

0  
77 29  
29  
  
  
  
---------------------------------------------------------------------------  
  
AttributeError Traceback (most recent call last)  
  
Input In [96], in <cell line: 13>()  
 11 print(iceTea.iceTeaSales,iceTea.temperature)  
 12 print(getattr(iceTea,'temperature'))  
---> 13 iceTea.price  
  
  
AttributeError: 'IceTeaSales' object has no attribute 'price'

\_\_slots\_\_阻止了\_\_dict\_\_对象分配给实例，因此iceTea.\_\_dict\_\_会引发异常，提示实例化对象没有属性\_\_dict\_。

iceTea.\_\_dict\_\_

---------------------------------------------------------------------------  
  
AttributeError Traceback (most recent call last)  
  
Input In [90], in <cell line: 1>()  
----> 1 iceTea.\_\_dict\_\_  
  
  
AttributeError: 'IceTeaSales' object has no attribute '\_\_dict\_\_'

dir()收集整个类树中所有继承的名称。

print(dir(iceTea))  
print('temperature' in dir(iceTea))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'iceTeaSales', 'temperature']  
True

\_\_init\_\_构造方法初始化参数，如果参数名不在\_\_slots\_\_列表中，也会引发异常。

class IceTeaSales:  
 \_\_slots\_\_=['temperature','iceTeaSales']  
 def \_\_init\_\_(self):  
 self.temperature=0  
 self.iceTeaSales=0  
 self.price=0  
iceTea=IceTeaSales()

---------------------------------------------------------------------------  
  
AttributeError Traceback (most recent call last)  
  
Input In [97], in <cell line: 7>()  
 5 self.iceTeaSales=0  
 6 self.price=0  
----> 7 iceTea=IceTeaSales()  
  
  
Input In [97], in IceTeaSales.\_\_init\_\_(self)  
 4 self.temperature=0  
 5 self.iceTeaSales=0  
----> 6 self.price=0  
  
  
AttributeError: 'IceTeaSales' object has no attribute 'price'

如果在\_\_slots\_\_列表中包含\_\_dict\_\_，则可以增加新的属性名，\_\_dict\_\_则会包含非\_\_slots\_\_列表中新增加的属性名键值对。

class IceTeaSales:  
 \_\_slots\_\_=['temperature','iceTeaSales', '\_\_dict\_\_']  
 def \_\_init\_\_(self):  
 self.temperature=0  
 self.iceTeaSales=0  
 self.price=0  
iceTea=IceTeaSales()  
print(iceTea.price)  
iceTea.name='flower tea'  
print(iceTea.name)  
print(iceTea.\_\_slots\_\_)  
print(iceTea.\_\_dict\_\_)

0  
flower tea  
['temperature', 'iceTeaSales', '\_\_dict\_\_']  
{'price': 0, 'name': 'flower tea'}

* Slot应用规则：

如果存在子类，在用\_\_slots\_\_时则需要注意：1. 子类中有\_\_slots\_\_，但父类中未配置\_\_slots\_\_，则实例对象总可以访问\_\_dict\_\_属性，因此没有意义。父类中有\_\_slots\_\_，而子类没有，同上，也没有意义；2. 子类定义了与父类相同的\_\_slots\_\_， 只能从父类中的\_\_slots\_\_获取定义的属性名。

class C:pass  
class D(C):\_\_slots\_\_=['a']  
  
X=D()  
X.a=1;X.b=2  
print(X.\_\_dict\_\_)  
print(D.\_\_dict\_\_.keys())

{'b': 2}  
dict\_keys(['\_\_module\_\_', '\_\_slots\_\_', 'a', '\_\_doc\_\_'])

* 内存使用量测试

使用[memory-profiler](https://pypi.org/project/memory-profiler/)④库，测量代码内存的使用率。该模块对python程序的内存消耗进行逐行分析，从而监控一个进程的内存消耗，该模块依赖[psutil](https://pypi.org/project/psutil/)⑤库。

从计算结果来看，未使用\_\_slots\_\_，内存变化为16.7MiB; 使用\_\_slots\_\_，内存变化为5.8MiB，因此使用\_\_slots\_\_可以有效节约内存空间。

JupyterLab中无法执行，需要在Spyder中运行（保存为模块）

**未使用\_\_slots\_\_:**

from memory\_profiler import profile  
  
class A(object):   
 def \_\_init\_\_(self,x):  
 self.x=x  
   
@profile  
def main():  
 f=[A(523825) for i in range(100000)]  
   
if \_\_name\_\_=='\_\_main\_\_':  
 main()

# Line # Mem usage Increment Occurences Line Contents

7 142.2 MiB 142.2 MiB 1 @profile  
 8 def main():  
 9 158.9 MiB 16.7 MiB 100003 f=[A(523825) for i in range(100000)]

**使用\_\_slots\_\_:**

from memory\_profiler import profile  
   
class A(object):  
 \_\_slots\_\_=('x')  
 def \_\_init\_\_(self,x):  
 self.x=x   
   
@profile  
def main():  
 f=[A(523825) for i in range(100000)]  
   
if \_\_name\_\_=='\_\_main\_\_':  
 main()

# Line # Mem usage Increment Occurences Line Contents

12 142.1 MiB 142.1 MiB 1 @profile  
13 def main():  
14 147.9 MiB 5.8 MiB 100003 f=[A(523825) for i in range(100000)]

注释（Notes）：

① functools，用于高阶函数：作用于或返回其他函数的函数，通常，任何可调用对象都可以视为函数，（<https://docs.python.org/3/library/functools.html>）。

② decorator，模块的目标是使定义保留签名（signature-preserving）的函数装饰器和装饰器工厂（factories）变得容易，（<https://pypi.org/project/decorator/>）。

③ abc，提供了在 Python 中定义抽象基类 (ABC) 的基础结构，（<https://docs.python.org/3/library/abc.html>）。

④ memory-profiler，用于监视进程的内存消耗及对 Python 程序内存消耗进行逐行分析。注意，该库已不再维护，（<https://pypi.org/project/memory-profiler/>）。

⑤ psutil，用于在 Python 中检索有关正在运行的进程和系统利用率（CPU、内存、磁盘、网络、传感器）的信息。 它主要用于系统监视、分析和限制进程资源以及运行进程的管理。，（<https://pypi.org/project/psutil/>）。

参考文献（References）:

[1] Primer on Python Decorators, <https://realpython.com/primer-on-python-decorators/>.

[2] classmethod() in Python, <https://www.geeksforgeeks.org/classmethod-in-python/>.

PC9-ipynb download