# string常用操作

## 删除字符串中多个指定字符

方法一：循环遍历字符，调用str.replace进行替换

chars = ",;:.! \t"  
while True:  
 s1 = input('Enter a string: ')  
 s1 = s1.strip()  
 if s1.lower() == 'quit':  
 print('QUIT!')  
 break  
 for c in chars:  
 s1 = s1.replace(c, '')  
 s2 = s1[::-1]  
 if s1 == s2:  
 print("Yes, it is a palindrome")  
 else:  
 print("No, it is not a palindrome")

方法二：调用re.sub方法进行正则表达式替换

import re  
  
chars = ",;:.! \t"  
while True:  
 s1 = input('Enter a string: ')  
 s1 = s1.strip()  
 if s1.lower() == 'quit':  
 print('QUIT!')  
 break  
 s2 = re.sub(chars, '', s1)  
 if s1 == s2:  
 print("Yes, it is a palindrome")  
 else:  
 print("No, it is not a palindrome")

方法三：调用str.makestrans设置转标换，str.translate进行字符转换

chars = ",;:.! \t"  
chars\_map = dict()  
for c in chars:  
 chars\_map[c] = None  
print(chars\_map)  
trans\_table = str.maketrans(chars\_map)  
  
while True:  
 s1 = input('Enter a string: ')  
 s1 = s1.strip()  
 if s1.lower() == 'quit':  
 print('QUIT!')  
 break  
 s2 = s1.translate(trans\_table)  
 if s1 == s2:  
 print("Yes, it is a palindrome")  
 else:  
 print("No, it is not a palindrome")

# 专题讨论

## 装饰器decorator

### 不带参数的decorator（二层嵌套def）

关键知识点：

* 通用化的参数传递(\*args, \*\*kwargs)
* 通过 @functools.wraps保存函数的\_\_name\_\_和\_\_doc\_\_属性

# -\*- coding:utf-8 -\*-  
  
import functools  
  
  
# def increment(func):  
# # wrapped\_f(a, b)是固定的参数传递方法，仅接受指定格式的参数形式  
# def wrapped\_f(a, b):  
# """ Increment a function result """  
# return func(a, b) + 1  
# return wrapped\_f  
  
  
def increment(func):  
 @functools.wraps(func)  
 # wrapped\_f(\*args, \*\*kwargs)是更通用的参数传递方法，可以任意的参数形式  
 def wrapped\_f(\*args, \*\*kwargs):  
 *""" Increment a function result """* return int(func(\*args, \*\*kwargs) + 1)  
 return wrapped\_f  
  
  
@increment  
def test\_plus(a, b):  
 *""" Add two things together """* return a + b  
  
  
# 如果设置@functools.wraps(func), 会负责把原始函数的\_\_name\_\_和\_\_doc\_\_等属性复制到wrapped\_f()函数中。  
print(test\_plus.\_\_name\_\_) # this returns 'test\_plus'  
print(test\_plus.\_\_doc\_\_) # this returns 'Add two things together'  
  
  
# 如果缺少@functools.wraps(func)  
# print(test\_plus.\_\_name\_\_) # this is now 'wrapped\_f' instead of 'test\_plus'  
# print(test\_plus.\_\_doc\_\_) # this now returns 'Increment a function result' instead of 'Add two things  
  
  
r = test\_plus(5, 5)  
assert r == 11, "We wrote our decorator wrong!"

### 携带参数的decorator（三层嵌套def）

def log(level):  
 def log\_decorator(func):  
 @functools.wraps(func)  
 def wrapped\_f(\*args, \*\*kwargs):  
 print('{0} logging info ...'.format(level))  
 return func(\*args, \*\*kwargs)  
 return wrapped\_f  
 return log\_decorator  
  
  
# 翻译成高阶函数的调用: test\_multiply = log('ERROR')(test\_multiply)  
@log('ERROR')  
def test\_multiply(a, b):  
 return a \* b  
  
  
s = test\_multiply(5, 5)  
assert s == 25

把装饰过程进行拆解后，过程如下：

log\_tmp = log('ERROR') # step1: 返回一个decorator函数  
  
  
# step2: 用log\_tmp去装饰test\_multiply并返回新函数  
@log\_tmp  
def test\_multiply(a, b):  
 return a \* b  
  
  
s = test\_multiply(5, 5)  
assert s == 25

考虑把三层嵌套的decorator定义拆分定义，此时最内层的wrapped函数引用了未定义的参数level，导致运行异常。

def log\_decorator(func):  
 @functools.wraps(func)  
 def wrapped\_f(\*args, \*\*kwargs):  
 # \*\*\* 此处引用了未定义的level，调用失败 \*\*\*  
 print('{0} logging info ...'.format(level))  
 return func(\*args, \*\*kwargs)  
 return wrapped\_f  
  
  
def log(level):  
 return log\_decorator  
  
  
@log('ERROR')  
def test\_func(a, b):  
 return a + b  
  
  
test = log('ERROR')(test\_func)  
a = test\_func(10, 10)  
assert a == 20

# log\_decorator运行异常，提示level变量未定义

Traceback (most recent call last):

File "D:/python-study/Hello/decorator.py", line 94, in <module>

a = test\_func(10, 10)

File "D:/python-study/Hello/decorator.py", line 79, in wrapped\_f

print('{0} logging info ...'.format(level))

NameError: name 'level' is not defined

## 字符串编码问题

**ASCII编码：**1字节编码，只有127个英文字符。

**Unicode编码：**一个不断发展的标准，通常是2个字节编码（非常偏僻的字符需要4个字节编码）。

--- 问题：如果统一成Unicode编码，乱码问题从此消失了。但是，如果文本内容基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

**UTF-8编码：**把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。

注意：

* Python2.X源码文件默认使用ascii编码（sys.getdefaultencoding() == ascii），默认不可以正常解析中文，源文件需要指定UTF-8编码。
* Python3.X 源码文件默认使用utf-8编码（sys.getdefaultencoding() == utf-8），所以可以正常解析中文，源文件无需指定 UTF-8 编码。

# FAQs

## ？？？浅拷贝copy和深拷贝deepcopy

## hashable和immutable的关系？

# 来自stackoverflow的高票数回答：

[**Hashing**](http://en.wikipedia.org/wiki/Hash_function) is the process of **converting some large amount of data into a much smaller amount (typically a single integer) in a repeatable way** so that it can be looked up in a table in constant-time (O(1)), which is important for high-performance algorithms and data structures.

[**Immutability**](http://en.wikipedia.org/wiki/Immutable_object) is the idea that an object **will not change in some important way after it has been created**, especially in any way that might change the hash value of that object.

The two ideas are related because **objects which are used as hash keys must typically be immutable so their hash value doesn't change.** If it was allowed to change then the location of that object in a data structure such as a hashtable would change and then the whole purpose of hashing for efficiency is defeated.