**回文字串的判断**

回文串表示的是将字符串逆序之后还能得到自身的字串。例如abcdcba逆序之后仍然是abcdcba。这样的字符串具有一个规律，即两边的字符一定相等，并且向中心点收敛的过程中左右两个边界字符仍保持相等。

解法：暴力判断法：从左右边界开始向中心收敛，如果出现了某一个位置左右边界不同则不是。

def is\_palindrome(s):

if len(s) <= 1:

return True

left, right = 0, len(s)-1

while left < right:

if s[left] != s[right]:

return False

left += 1

right -= 1

return True

扩展：如果仅仅判断字符串是否回文的问题是简单的，但是往往存在一些深入的问题，例如求解最长回文子串和最短回文子串，还有一些字符拼接和填充字符生成回文字串的问题。

对于这些问题，往往是属于求解符合题意的特定或所有回文字串。

回文字串具有一个规律，就是会有一个单字符或者空的中心点，然后以这个中心向两边扩展的字符均相等。这也是判断回文的重要依据，此外通过向外扩展的方式也可以轻松得到回文子串的长度。

以求解最长回文串为例，可以遍历所有中心点，由于含有中心可能为空的情况，所以总的情况数为2\*n-2，假设长度为n。遍历范围[0,2\*n-2]且中心根据奇偶性判断，如果是奇数则中心是i//2,否则中心为空，确定左右边界为i//2和i//2+1，向两边扩展判断并统计长度，维护最长回文串的解。

解法：暴力搜索法：遍历字符串每一个位置，判断其作为中心是否可以得到回文串以及确定回文串的长度。

还有一类题是填充或拼接，例如给定一个字符串，求解在字符串前添加字符使得该字符串为回文字串最少所需要的添加的字符个数。

思路：对于这个题，首先注意到是只能在字符串前加字符，也就是说如果字符串前缀中有回文子串，那么可以忽略，因为构成回文串的时候前缀的字符串一定是在中心位置。那么添加最少字符就转换成了寻找字符串的最大前缀回文子串的问题，只要确定了前缀串，添加剩余字串的逆序到前面即可。

解法1：暴力法：遍历字符，求解每个字符作为中心位置可以得到的最长回文子串，而且左侧索引必须到达0才是前缀字符串。或者计算以每个字符结尾的最长回文字串，这样直接从两边开始判断向中心收敛，如果得到是回文子串，计算索引差即可得到长度。维护最长的结果即可。时间复杂度为O(n^2)

def get\_length(pos):

# 求奇数回文长度

length\_odd = 1

l = pos - 1

r = pos + 1

while l >= 0 and r < len(s) and s[l] == s[r]:

length\_odd += 2

l -= 1

r += 1

if l != -1:

length\_odd = 1

# 求偶数回文长度

length\_even = 0

l = pos

r = pos + 1

while l >= 0 and r < len(s) and s[l] == s[r]:

length\_even += 2

l -= 1

r += 1

if l != -1:

length\_even = 0

if length\_odd >= length\_even:

return length\_odd, True

else:

return length\_even, False

解法2：Rabin-Karp算法，即字符串编码算法。将字符串看作一个base进制的数，对应的十进制就是得到hash数，在对模数取模之后，如果得到相同的hash值则说明两个字符串相等。而对于hash冲突在这里概率非常低可以忽略，但在工程项目中存在如此可能所以要避免，会造成系统bug。对于本题对一个字符串及其逆序求hash值，如果相同则说明正序逆序串相等，所以是一个回文字串。

n = len(s)

last = -1

base, MOD = 131, 10 \*\* 9 + 7

mul = 1

# left和right分别表示当前字符结尾的字符串正序和逆序对应的hash值，如果两个相等则表示当前字符串是回文子串，因为一个字符串正序逆序相等是回文

left, right = 0, 0

for i in range(n):

left = (left \* base + ord(s[i])) % MOD

right = (right + ord(s[i]) \* mul) % MOD

if left == right:

last = i

mul = mul \* base % MOD

return s[::-1] + s[last+1:]

解法3：KMP算法。KMP算法用于求解字符串匹配问题，而在本题，利用回文字串的特性，将s逆序后得到s’，将s作为模式串而s’作为目标串，如果s可以匹配上s’则表示s本身是一个回文子串，而匹配不上的情况则可以根据s匹配结束时对齐的最后一位的位置求解最大长度的前缀串。因为此时的匹配刚好时s’的后缀对应s的前缀，而根据此前的分析，如果前缀和后缀匹配成功，则证明是回文字串。

for i in range(1, n):

j = fail[i - 1]

while j != -1 and s[j + 1] != s[i]:

j = fail[j]

if s[j + 1] == s[i]:

fail[i] = j + 1

pattern\_ptr, target\_ptr = 0, 0

s\_reverse = s[::-1]

while target\_ptr < n and pattern\_ptr < n:

if pattern\_ptr == -1 or s[pattern\_ptr] == s\_reverse[target\_ptr]:

target\_ptr += 1

pattern\_ptr += 1

else:

pattern\_ptr = fail[pattern\_ptr]

if pattern\_ptr == n:

return s

else:

return s\_reverse[:n-pattern\_ptr] + s

除此之外还有一种情况复杂的求next的方式

# pattern\_ptr = 0

# prefix\_ptr = -1

# # 求next数组，因为next是关键字，这里使用fail表示，就意义来说是匹配失败需要回溯的

# while pattern\_ptr < n - 1:

# if prefix\_ptr == -1 or s[pattern\_ptr] == s[prefix\_ptr]:

# # 当前相等要后移

# pattern\_ptr += 1

# prefix\_ptr += 1

# # 如果不同，则[0,prefix\_ptr-1]的字串是相等的，可以回退到prefix\_ptr的位置，从而跃进最大步

# if s[pattern\_ptr] != s[prefix\_ptr]:

# fail[pattern\_ptr] = prefix\_ptr

# # 如果当前模式串指向字符等于前缀指针指向的字符，那回退到prefix\_ptr的位置也不可能匹配，所以要回退到更早的位置

# else:

# fail[pattern\_ptr] = fail[prefix\_ptr]

# else:

# # 不相等的情况下表明后续更不可能出现匹配，将前缀指针前移

# prefix\_ptr = fail[prefix\_ptr]