

实验报告 ps4

【题意理解及思路】

Part A: Permutations of a string

求一个字符串的全排列，必须采用递归的方法实现

实现函数 `get_permutations(sequence)` 功能，返回字符串 `sequence` 的全排列。并给出 3 个案例来证明函数功能实现正确，其中每个案例需要符合题目给出的格式进行输出，长度为 n 的字符串的全排列有 $n!$ 个，因此，为了便于给出预测的输出 Expected Output，给出案例的字符串长度建议 小于等于 3

思路：

递归思想：

(1)、一个长度为 n 的字符串 `sequence`，从中取出一个元素 `sequence[i]`，其中 i 可以是字符串中任意位置元素，有 n 种可能；取出之后，剩下的字符串长度为 $n-1$ ，则原字符串 `sequence` 的全排列就为 `sequence[i]` + 剩下 $n-1$ 个元素组成的字符串的全排列。

(2)、递归截止条件：当字符串长度为 1，结果就是该字符串。

(3)、显然结果是有很多个字符串组成，因此用列表返回结果。

代码：

```
def get_permutations(sequence):
    # 长度为1 递归截止条件
    if len(sequence) == 1:
        return [sequence]

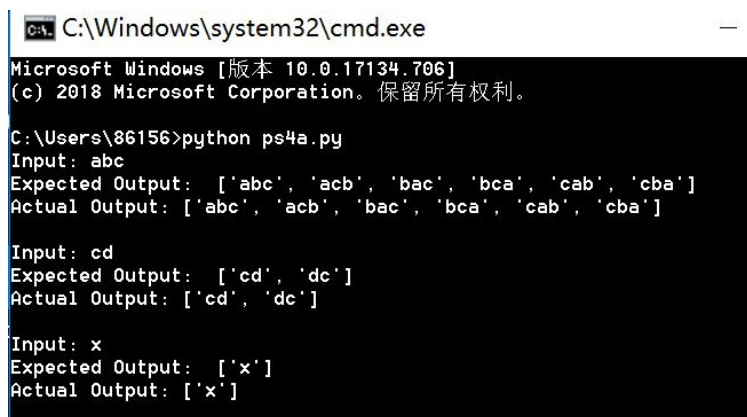
    result = [] # 储存返回的结果
    for i in range(len(sequence)):
        # 每次取出i位置元素
        string_temp = sequence[:i] + sequence[i+1:] # 剩下元素组成的新字符串
        # 剩下元素的全排列（列表）
        perm_after = get_permutations(string_temp)
        for x in perm_after:
            # 将取出的元素放到字符串最前面，再添加到结果返回列表中
            result.append(sequence[i] + x)
    return result
```

为了减少代码量，3 个案例及其正确的输出就用列表存起来了。但是最终的输出格式还

是符合题目要求的。

```
if __name__ == '__main__':
    case=3
    example_input = ['abc', 'cd', 'x']
    expected_output=[
        ['abc', 'acb', 'bac', 'bca', 'cab', 'cba'],
        ['cd', 'dc'],
        ['x']
    ]
    for i in range(case):
        print('Input:', example_input[i])
        print('Expected Output:', expected_output[i])
        print('Actual Output:', get_permutations(example_input[i]))
        print("")
```

运行结果：



```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.17134.706]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\86156>python ps4a.py
Input: abc
Expected Output: ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
Actual Output: ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

Input: cd
Expected Output: ['cd', 'dc']
Actual Output: ['cd', 'dc']

Input: x
Expected Output: ['x']
Actual Output: ['x']
```

Part B: Cipher Like Caesar

Part 1: Message

先理解一些基本概念的含义，如消息的加密、解密、密码、明文、密文。

按代码注释要求实现三个类中的方法 Message、PlaintextMessage 和

CiphertextMessage，其中后面两个类是继承自父类 Message，它们都是新式类。实现

后两个类中方法时有时需要你调用父类中的方法，

Caesar Cipher（凯撒密码），重点是理解 shift，build_shift_dict(self, shift)，apply_shift(self, shift) 这三者的含义与关系。其中，shift 是一个数字，原理虽然很简单，但是这里用文字表达起来比较绕口，写了一大段自己都没被说服。决定换种方式阐述原理。

一张图胜过千言万语：



如图所示,假如消息字符串中某一位字母是 A , shift 值为 6, 那么该消息经过编码 (或者说加密) 后, A 所在位置的字母就变成了 G, 同理, 如果原文该位置字母是 a, 则密文该字母是 g , shift 值是变量, 如果 shift 为 7 ,相应密文该位置就变成了 H。那么 shift 值可以理解为将圆盘逆时针旋转的 次数, 每旋转一次, 字母后移动一位。

build_shift_dict(self, shift): 因为有大小写共 52 个字母, 每一个字母都经过上面循环左移的处理, 就建立了一种映射关系, 如上面的例子中的 A 映射到 G, 那么 52 个字母每个字母在特定的 shift 值下都有唯一确定的映射值, 如何表示这种映射关系呢? 就用 build_shift_dict 函数, 返回一个字典, 存储着 52 个字母的原值和映射值, 在这里我们称它为加密字典。这个字典的建立只依赖于 shift 值, 一旦 shift 值确定, 对应的加密字典也确定了。因为是循环的, 也就是说, shift=0 和 shift=26 效果都是一样的。

#建立有符合加密规则的字母映射字典

```
def build_shift_dict(self, shift):  
    #字典键为原字母，键值为对应字母的加密之后的字母，大小写分开处理  
    lower_keys = list(string.ascii_lowercase)  
    lower_values = list(string.ascii_lowercase)  
    shift_lower_values = lower_values[shift:] + lower_values[:shift]  
  
    upper_keys = list(string.ascii_uppercase)  
    upper_values = list(string.ascii_uppercase)  
    upper_shift_values = upper_values[shift:] + upper_values[:shift]  
  
    total_keys = lower_keys + upper_keys  
    total_values = shift_lower_values + upper_shift_values  
  
    #大小写合并到一个字典  
    self.shift_dict = dict(zip(total_keys, total_values))  
    return self.shift_dict
```

apply_shift(self, shift): apply 应用，这个函数就是说将上面的处理过程应用到具体的消息对象中，也就是针对特定的 shift 值对消息中每一个字母（必须是字母，其它字符不做此处理）进行上面的“循环左移处理”。

这里不能单独将其翻译为加密过程，虽然它确实实现了加密功能。但是此时我们考虑解密（译码）过程，简单直接的想法，加密是圆盘逆时针转，解密只需要顺时针转相应的步数即可。我们总没必要为了解密再建立一个就像加密过程一样的映射字典吧。圆盘顺时针转 shift 次，其效果就相当于逆时针转 26-shift 次。那么我们在解密过程依然是通过使用 apply_shift(shift)这个函数，只是对应的 shift 要相应改变，所以在这里不将其翻译为加密处理过程，在代码中注释为编码过程。

#编码过程

```
def apply_shift(self, shift):  
    new_msg = [] #存储结果  
    for i in self.message_text:  
        #出现不是字母的字符，保持原样  
        if i not in self.build_shift_dict(shift).keys():  
            new_msg.append(i)  
            continue  
        else:  
            new_msg.append(self.build_shift_dict(shift)[i])  
    #返回处理后的消息  
    return ''.join(new_msg)
```

剩下的一些初始化和获取相关消息的方法就不在这里贴出来了, 代码下方的注释已经将实现过程解释的不能再清楚了。

Part 2: PlaintextMessage

首先, 了解 super 用法调用基类函数即可很容易实现这个类。

然后这里个人觉得值得注意的是 change_shift(self, shift)方法, 这里不单单只是个简单的重新赋值就可以了, 当 shift 改变之后, 对应的字母加密映射字典也会变, 加密之后的消息也会随之改变。代码如下:

```
def change_shift(self, shift):
    self.shift = shift
    #shift改变, 注意对应的加密映射字典也得改变, 加密消息也会变
    self.encrypting_dict = super(PlaintextMessage, self).build_shift_dict(shift)
    self.message_text_encrypted = super(PlaintextMessage, self).apply_shift(shift)
```

Part 3: CiphertextMessage

这部分主要需要实现的是一个对消息的解密方法。关键是找到这个最佳的 shift 值, 最佳的含义是指: 当我们对一个消息使用 apply_shift(shift)处理时, 使得解密后消息中有效单词数量最多的 shift 值。显然 shift 值取值有 26 种可能。因此采用两层 for 循环, 外层测试每一个可能的 shift 值, 内层统计并更新消息中有效单词的数量 word_counter 并及时更新最大有效单词数量 max_counter 值, 再将此时的 shift 值作为参数对消息进行解码。

```
#消息解密
def decrypt_message(self):
    word_counter = 0 #real words 的数量
    max_counter = 0 #real words 的数量最大值
    for i in range(26):#对每个shift可能判断
        for j in list(super(CiphertextMessage, self).apply_shift(i).split(' ')):
            if is_word(self.valid_words, j):
                word_counter += 1
            if word_counter > max_counter:
                #更新max_counter 并记录此时i值
                max_counter = word_counter
                shift_value = i
                decrypted_msg = super(CiphertextMessage, self).apply_shift(i)
    return (shift_value, decrypted_msg)
```


Part 4: Testing

对 PlaintextMessage 和 CiphertextMessage 对象分别写两个测试案例来验证功能。

输出的格式需要符合代码注释要求。另外将密文 story.text 解密处理, 并返回最佳 shift 值。

所给案例包含字母大小写, 特殊字符, 以及选择了类似 Y 这种比较靠后这种字母来体现循环移位的实现。

```
#TODO: best shift value and unencrypted story
plaintext = PlaintextMessage("Yellow!", 5)
print("Expected Output: Djqqtb!")
print("Actual Output:", plaintext.get_message_text_encrypted())
print("")

plaintext = PlaintextMessage("World!", 2)
print("Expected Output: Yqtnf!")
print("Actual Output:", plaintext.get_message_text_encrypted())
print("")

ciphertext = CiphertextMessage("Djqqtb!")
print("Expected Output:", (21, "Yellow!"))
print("Actual Output:", ciphertext.decrypt_message())
print("")

ciphertext = CiphertextMessage("Yqtnf!")
print("Expected Output:", (24, "World!"))
print("Actual Output:", ciphertext.decrypt_message())
print("")

#TODO: best shift value and unencrypted story
print(CiphertextMessage(get_story_string()).decrypt_message())
```

```
C:\Users\86156>python ps4b.py
Loading word list from file...
 55901 words loaded.
Expected Output: Djqqtb!
Actual Output: Djqqtb!

Loading word list from file...
 55901 words loaded.
Expected Output: Yqtnf!
Actual Output: Yqtnf!

Loading word list from file...
 55901 words loaded.
Expected Output: (21, 'Yellow!')
Actual Output: (21, 'Yellow!')

Loading word list from file...
 55901 words loaded.
Expected Output: (24, 'World!')
Actual Output: (24, 'World!')

Loading word list from file...
 55901 words loaded.
(22, 'Tkmu Puyboi sc k widrmku mrbbkmdob mbokdon yx dro czeb yp k wywoxd dy rou
z myfob kx sxeppsmsoxdvi zukxxon rkmu. Ro rkc loox boqscdobon pyb mukccoc kd WS
D dgsmo lopybo. led rkc bozybdonvi xofob zkccoon kmukcc. Sd rkc loox dro dbknsdsy
x yp dro boesnoxdc yp Okcd Mkwzec dy lomywo Tkmu Puyboi pyb k pog xsqrde okmr io
kb dy onemkdo sxmywsxq cdenoxdc sx dro gkic, wokxc, kxn odrsmc yp rkmusxq.')
C:\Users\86156>
```

至于 story.text 的解密验证，只能根据求出的 shift_value 值再去对比原文密文进行相关验证了，shift_value 没错的话，得到的明文可以确保是没错的。

Part C: Substitution Cipher

这个部分要解决的问题还是一样，消息的加密和解密，只是规则有所变化。同样，首先根据代码注释实现一些初始化和获取一些属性的方法。然后，就像上面 partb 中一样的思想，我们首先需要建立一种映射关系，来表示消息中某个位置上字母在加密前后的对应关系。

也就是 build_transpose_dict 这个函数需要实现的功能。显然这个函数返回值应该是个字典，来存储具有上述映射关系的键值对。字典的键即为 52 个字母。而加密的规则便是，辅音字母不变，其键和值一样，元音字母则被替换为 5 个元音字母的某一个特定排列的相同位置的字母，也就是说，我们首先需要获取到元音字母的一个排列，一旦这个排列确定了，对于一个消息字符串中的所有元音字母，都将按该对应关系进行映射。那么这个函数的参数就应该是元音字母的一个排列。

```
# 建立映射字典
def build_transpose_dict(self, vowels_permutation):
    # 字典键 原文字母
    plain = VOWELS_LOWER + VOWELS_UPPER + CONSONANTS_LOWER + CONSONANTS_UPPER
    # 字典值 密文对应位置的字母
    cipher = vowels_permutation.lower() + vowels_permutation.upper() + CONSONANTS_LOWER + CONSONANTS_UPPER
    return {plain[i]: cipher[i] for i in range(len(plain))}
```

Part 2: EncryptedSubMessage

加密规则确定了，接下来就是将该规则运用到具体的消息中，也就是函数 apply_transpose 需要实现的功能。当然这个函数的功能不只是加密，就像之前介绍过的 apply_shift 一样，它们都只是一个工具，将某个字母映射为另一个字母，而将另一个字母映射回原字母也是用它，所以加密解密都需要用它。只是在特定情况下需要理解为加密，特

定情况下理解为解密。只需要将明文中每个字符作为字典键，用循环逐个取键值即可实现：

```
#加密消息
def apply_transpose(self, transpose_dict):
    text = self.message_text #原文
    ciphertext = [] #密文
    #字符逐个映射
    for ch in text:
        #返回指定键的值 不存在则返回key本身
        ciphertext.append(transpose_dict.get(ch,ch))
    return "".join(ciphertext)
```

完善 EncryptedSubMessage 类，这个类继承自 SubMessage。

其实际完成的功能是一个解密的过程。实现过程中需要调用父类中的方法，准确的说是
要很灵活的调用方法。

思想跟上面一样，采用双层循环，对元音字母的每一个排列，求相应情况下解密之后有
效单词的数量，在循环中更新有效单词数量最大值 max_counter 并进行解密处理。

```
def decrypt_message(self):
    #获取全排列
    perms = get_permutations(VOWELS_LOWER)
    max_counter = 0 #最大有效单词数量
    decrypt_msg = "" #解密后明文
    for perm in perms:
        temp_dict = self.build_transpose_dict(perm)
        temp_text = self.apply_transpose(temp_dict).split()
        counter=0 #计算有效单词数量
        for j in temp_text:
            if is_word(self.valid_words, j):
                counter += 1
        #更新max_counter值 并且更新解密后明文
        if counter > max_counter:
            max_counter = counter

        decrypt_msg = self.apply_transpose(temp_dict)
    return decrypt_msg
```

Part 3: Testing

对加密和解密过程分别写两个测试案例，输出最佳解密效果，所谓最佳，是指解密之后
的消息字符串中有效单词数量最多的情况下的解密，输出格式和代码给出的 example 要求

一致。代码这里就不贴了，直接赋值示例代码，再修改 message 值和 permutation 值。

为使得案例具有代表性，给出的原文含有特殊字符，尽可能多的元音字符，不同 permutation 时的验证。结果如下：

```
C:\Users\86156>python ps4c.py
Loading word list from file...
55901 words loaded.
Original message: Hello World! Permutation: eaiuo
Expected encryption: Hallu Wurld!
Actual encryption: Hallu Wurld!
Loading word list from file...
55901 words loaded.
Decrypted message: Hello World!

Loading word list from file...
55901 words loaded.
Original message: Quite easy! Permutation: eaiuo
Expected encryption: Quita aosy!
Actual encryption: Qoita aesy!
Loading word list from file...
55901 words loaded.
Decrypted message: Quite easy!

Loading word list from file...
55901 words loaded.
Original message: Follow your heart! Permutation: uoiea
Expected encryption: Fellow year hourt!
Actual encryption: Fellow year hourt!
Loading word list from file...
55901 words loaded.
Decrypted message: Follow your heart!

C:\Users\86156>
```

【实验心得】

这次的问题，主要是考察面向对象编程，因为学过 C++ 的缘故，实现起来难度不是很大，而递归实现全排列，应该是以前就实现过的，只是参数形式稍有变化，写完全排列，对比 C++，再次感受到 python 语法的灵活飘逸。

而完善类的方法，则在题意的理解上需要花费一些功夫，其实真正需要实现功能的方法不多，主要是加密和解密，剩下的一些初始化方法和获取信息的方法基本代码下方的注释已经给出八九成了。需要对类和继承一些类的方法有些基本掌握，比如 super 的用法。

另一个解决问题的重点是用好函数的返回值，函数的返回结果作为另一个函数的参数，灵活调用可以省去很多代码。

最后字符串列表操作依然是 python 的重头戏，常用字符串列表操作需要做到精通。

