第五题

第五题包含若干小问，以下以（一）、（二）、（三）等标记每一小问。

（一） 修改 smith_waterman()函数，使用替换矩阵，而不是匹配或不匹配参数。

这段代码实际上是对第三题中 smith_waterman()函数的修改，对比这两段代码，发现修改的部分主要是在对 score1 的赋值上。

下面介绍主要的区别：

原先的代码使用 if……else 判断，如果待比较的两个字符相等，则加 match 的值，否则（不等）则在 matrix[I,j]后加 mismatch 的值。而现在的代码，则无判断，直接在其后添加键“a1+a2”在字典 submat 的对应的值。为什么可以这样实现？我们回顾一下第四题，第四题主要构建了一个“替换矩阵”的字典，该字典的键为“给定两字符”，而值的赋值是如果该两字符相等，则将 match 的值赋上；否则是 mismatch 的值；其实这个过程就实现了之前 if……else 的那个过程。而序列比对本质上来讲也就是比较特定的位置上两字符匹配或不匹配。这样做有什么好处呢？我觉得是使代码的灵活性更强。我们直观上可以看到，修改后的代码比之前的代码更加的简洁；替换矩阵的字典构建好之后，在接下来任何比对的过程中，我们都可以使用这个已经构建好的字典，一劳永逸，不需要再进行重复繁琐的判断两字符是否相等的过程。

```
11      for i,a1 in enumerate(s1):
12          for j,a2 in enumerate(s2):
13              score1 = matrix[i,j] + submat[a1+a2]
```

（这道题的代码修改）

```
11      for i,a1 in enumerate(s1):
12          for j,a2 in enumerate(s2):
13              if a1==a2:
14                  score1 = matrix[i,j] + match
15              else:
16                  score1 = matrix[i,j] + mismatch
```

（第三题的原始的 smith_waterman 函数 score1 的赋值）

最后的这个测试代码中，引用了三个函数。creat_substitution_matrix()是第四题中构建的函数，swith_waterman()为自第三题修改的同名函数，而 local_traceback()即为第三题同名的函数。

```
#Test Algorithm
s1 = "TCCCAGTTATGTCAGGGGACACGAGCATGCAGAGA"
s2 = "AATTGCCGCCGTCGTTTTCAGCAGTTATGTCAGAT"
alphabet = ["A","T","G","C"]
submat_dna = create_substitution_matrix(alphabet,5,-4)
score,matrix,traceback = smith_waterman(s1,s2,
                    submat_dna,gap=-1)
print("Optimal Score: %d" % score)
local_traceback(s1,s2,matrix,traceback) #Code siehe Übung 4
```

```
Optimal Score: 90
T----CC--CAGTTATGTCAGGGGACACG--A-GCATGCAGA
TGCCGCCGTC-GTT-T-TCA---G-CA-GTTATG--T-CAGA
```

（二） 导入 PAM250/PAM70 替换矩阵，重新进行比对。

首先需要明白这里的 PAM250/PAM70 的替换矩阵指的是什么？我们应该不陌生替换矩阵。所谓的替换矩阵就是当我们进行序列比对的时候，A 碱基（氨基酸）比对到 B 碱基（氨基酸）时，如果相同，加多少；如果不同，罚分多少。我们在第四题的时候，曾试图构建过一个简单的替换矩阵，如果两碱基（氨基酸）相等，则比对的时候赋值为 match 的分值，否则为 mismatch 的分值。

$$\text{Match} = 5$$
$$\text{Mismatch} = -4$$

	A	T	G	C
A	5	-4	-4	-4
T	-4	5	-4	-4
G	-4	-4	5	-4
C	-4	-4	-4	5

但是，在实际情况（自然界）下，并没有那么的简单，比如自然界中更容忍某一碱基替换成某一碱基，而相对的如果替换成另一碱基，则会发生致命的突变，该物种在漫长的演化过程中就会被淘汰。所以为了使我们的比对结果更加的可靠，

更加符合生物学的演化规律，我们需要参考自然界中真实的序列来构建替换矩阵。PAM250（PAM70）矩阵就是一种参考真实的数据构建的氨基酸替换矩阵。PAM250 是由 PAM1 矩阵进行“自乘”250 次得到的。PAM1 矩阵最初是由序列相似性相差 1%的氨基酸序列进行多序列比对计算，仅通过一次变异，氨基酸之间替换的概率。而对于序列相差较大的序列，PAM-1 这种矩阵变不再适用，因为两条序列之间可能存在若干次的变异，因此为了更好的模拟这种情况，将 PAM-1 矩阵自乘 n 次，模拟 A→……→B 氨基酸的变异过程，得到的就是两氨基酸在经过若干次的演化之后，变异成这样的一个结果的概率。

The PAM250 scoring matrix

.	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	2	-2	0	0	-2	0	0	1	-1	-1	-2	-1	-1	-3	1	1	1	-6	-3	0	0	0	0	-8
R	-2	6	0	-1	-4	1	-1	-3	2	-2	-3	3	0	-4	0	0	-1	2	-4	-2	-1	0	-1	-8
N	0	0	2	2	-4	1	1	0	2	-2	-3	1	-2	-3	0	1	0	-4	-2	-2	2	1	0	-8
D	0	-1	2	4	-5	2	3	1	1	-2	-4	0	-3	-6	-1	0	0	-7	-4	-2	3	3	-1	-8
C	-2	-4	-4	-5	12	-5	-5	-3	-3	-2	-6	-5	-5	-4	-3	0	-2	-8	0	-2	-4	-5	-3	-8
Q	0	1	1	2	-5	4	2	-1	3	-2	-2	1	-1	-5	0	-1	-1	-5	-4	-2	1	3	-1	-8
E	0	-1	1	3	-5	2	4	0	1	-2	-3	0	-2	-5	-1	0	0	-7	-4	-2	3	3	-1	-8
G	1	-3	0	1	-3	-1	0	5	-2	-3	-4	-2	-3	-5	0	1	0	-7	-5	-1	0	0	-1	-8
H	-1	2	2	1	-3	3	1	-2	6	-2	-2	0	-2	-2	0	-1	-1	-3	0	-2	1	2	-1	-8
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5	2	-2	2	1	-2	-1	0	-5	-1	4	-2	-2	-1	-8
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6	-3	4	2	-3	-3	-2	-2	-1	2	-3	-3	-1	-8
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5	0	-5	-1	0	0	-3	-4	-2	1	0	-1	-8
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6	0	-2	-2	-1	-4	-2	2	-2	-2	-1	-8
F	-3	-4	-3	-6	-4	-5	-5	-5	-2	1	2	-5	0	9	-5	-3	-3	0	7	-1	-4	-5	-2	-8
P	1	0	0	-1	-3	0	-1	0	0	-2	-3	-1	-2	-5	6	1	0	-6	-5	-1	-1	0	-1	-8
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2	1	-2	-3	-1	0	0	0	-8
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3	-5	-3	0	0	-1	0	-8
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17	0	-6	-5	-6	-4	-8
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	-2	-3	-4	-2	-8
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4	-2	-2	-1	-8
B	0	-1	2	3	-4	1	3	0	1	-2	-3	1	-2	-4	-1	0	0	-5	-3	-2	3	2	-1	-8
Z	0	0	1	3	-5	3	3	0	2	-2	-3	0	-2	-5	0	0	-1	-6	-4	-2	2	3	-1	-8
X	0	-1	0	-1	-3	-1	-1	-1	-	1	-1	-1	-1	-2	-1	0	0	-4	-2	-1	-1	-1	-1	-8
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1
.	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*

PAM70 则是 PAM1 仅自乘了 70 次，因此，这两个打分矩阵适用于不同的相似性特征的序列。

PAM 矩阵	PAM250	PAM70
适用情况	序列间变异大的两条序列	序列间变异小的两条序列

注：在生物信息学中常用的打分矩阵还有 BLOSUM 矩阵，由于时间原因，此不赘述。

这里代码又是如何实现的呢?

他主要做了什么事情呢？读取目录下的 PAM 打分矩阵的文件（就是你发的 PAM250.txt 的这个文件)。然后从中进行信息的提取，知道谁突变成谁的打分是多少，构建一个我们熟悉的字典的数据结构来存储这部分的信息，即为替换矩阵。

```
def load_substitution_matrix(filename):
    submat = {}
    f = open(filename,"r")
    for i,line in enumerate(f):
        if i==0:
            tokens_header = line.strip().split(",")[1:]
        else:
            a1 = line[0]
            scores = line.strip().split(",")[1:]
            for j,score in enumerate(scores):
                submat[a1 + tokens_header[j]] = float(score)
    f.close()
    return submat
```

我们先熟悉一下 PAM250.txt 的文件的基本的内容。

它的第一行和第一列是氨基酸名字的缩写。而中间的这个数值指的是，对应（行、列氨基酸）的替换的打分。如 A—>R,打分为-2；而 R→R,则打分为 6；(从进化上讲，打分越高，变换约容易发生，越保守。而打分为负，且分值很负，说明在进化上，该碱基间的替换的发生所需要的成本很大;)

```
#,A,R,N,D,C,Q,E,G,H,I,L,K,M,F,P,S,T,W,Y,V,B,Z,X,*
A,2,-2,0,0,-2,0,0,1,-1,-1,-2,-1,-1,-3,1,1,1,-6,-3,0,0,0,0,-8
R,-2,6,0,-1,-4,1,-1,-3,2,-2,-3,3,0,-4,0,0,-1,2,-4,-2,-1,0,-1,-8
N,0,0,2,2,-4,1,1,0,2,-2,-3,1,-2,-3,0,1,0,-4,-2,-2,2,1,0,-8
D,0,-1,2,4,-5,2,3,1,1,-2,-4,0,-3,-6,-1,0,0,-7,-4,-2,3,3,-1,-8
C,-2,-4,-4,-5,12,-5,-5,-3,-3,-2,-6,-5,-5,-4,-3,0,-2,-8,0,-2,-4,-5,-3,-8
Q,0,1,1,2,-5,4,2,-1,3,-2,-2,1,-1,-5,0,-1,-1,-5,-4,-2,1,3,-1,-8
E,0,-1,1,3,-5,2,4,0,1,-2,-3,0,-2,-5,-1,0,0,-7,-4,-2,3,3,-1,-8
G,1,-3,0,1,-3,-1,0,5,-2,-3,-4,-2,-3,-5,0,1,0,-7,-5,-1,0,0,-1,-8
H,-1,2,2,1,-3,3,1,-2,6,-2,-2,0,-2,-2,0,-1,-1,-3,0,-2,1,2,-1,-8
I,-1,-2,-2,-2,-2,-2,-2,-3,-2,5,2,-2,2,1,-2,-1,0,-5,-1,4,-2,-2,-1,-8
L,-2,-3,-3,-4,-6,-2,-3,-4,-2,2,6,-3,4,2,-3,-3,-2,-2,-1,2,-3,-3,-1,-8
K,-1,3,1,0,-5,1,0,-2,0,-2,-3,5,0,-5,-1,0,0,-3,-4,-2,1,0,-1,-8
M,-1,0,-2,-3,-5,-1,-2,-3,-2,2,4,0,6,0,-2,-2,-1,-4,-2,2,-2,-2,-1,-8
F,-3,-4,-3,-6,-4,-5,-5,-5,-2,1,2,-5,0,9,-5,-3,-3,0,7,-1,-4,-5,-2,-8
P,1,0,0,-1,-3,0,-1,0,0,-2,-3,-1,-2,-5,6,1,0,-6,-5,-1,-1,0,-1,-8
S,1,0,1,0,0,-1,0,1,-1,-1,-3,0,-2,-3,1,2,1,-2,-3,-1,0,0,0,-8
T,1,-1,0,0,-2,-1,0,0,-1,0,-2,0,-1,-3,0,1,3,-5,-3,0,0,-1,0,-8
W,-6,2,-4,-7,-8,-5,-7,-7,-3,-5,-2,-3,-4,0,-6,-2,-5,17,0,-6,-5,-6,-4,-8
Y,-3,-4,-2,-4,0,-4,-4,-5,0,-1,-1,-4,-2,7,-5,-3,-3,0,10,-2,-3,-4,-2,-8
V,0,-2,-2,-2,-2,-2,-2,-1,-2,4,2,-2,2,-1,-1,-1,0,-6,-2,4,-2,-2,-1,-8
B,0,-1,2,3,-4,1,3,0,1,-2,-3,1,-2,-4,-1,0,0,-5,-3,-2,3,2,-1,-8
Z,0,0,1,3,-5,3,3,0,2,-2,-3,0,-2,-5,0,0,-1,-6,-4,-2,2,3,-1,-8
X,0,-1,0,-1,-3,-1,-1,-1,-1,-1,-1,-1,-1,-2,-1,0,0,-4,-2,-1,-1,-1,-1,-8
*,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,-8,1
```

下面正式介绍代码的细节：

首先作者同样使用关键字 def 定义了名为“load_substitution_matrix”的函数。取输入的变量为文件名。

```
def load_substitution_matrix(filename):
```

在函数体中，作者首先声明了一个字典型的变量 submat，并初始化为空。

```
submat = {}
```

然后接下来，作者以只读的方式（**“r”**）读取了该文件，并将其命名为变量 f。

```
f = open(filename,"r")
```

注：在 Python 中，有三种读取文件的基本模式（还有其它）。

（1） r (r+) :读

（2） w (w+) ： 写

（3） a (a+) ： 追加

我们看一下，f 变量长什么样？（同样，当你不知道这样代码什么意思的时候，最直接的方法，就是用 print()输出，看看是什么？

```
>>> f = open("PAM250.txt","r")
>>> f
<_io.TextIOWrapper name='PAM250.txt' mode='r' encoding='UTF-8'>
>>> for i,line in enumerate(f):
...     print(i)
...     print(line)
...
0
#,A,R,N,D,C,Q,E,G,H,I,L,K,M,F,P,S,T,W,Y,V,B,Z,X,*

1
A,2,-2,0,0,-2,0,0,1,-1,-1,-2,-1,-1,-3,1,1,1,-6,-3,0,0,0,0,-8

2
R,-2,6,0,-1,-4,1,-1,-3,2,-2,-3,3,0,-4,0,0,-1,2,-4,-2,-1,0,-1,-8

3
N,0,0,2,2,-4,1,1,0,2,-2,-3,1,-2,-3,0,1,0,-4,-2,-2,2,1,0,-8

4
D,0,-1,2,4,-5,2,3,1,1,-2,-4,0,-3,-6,-1,0,0,-7,-4,-2,3,3,-1,-8

5
C,-2,-4,-4,-5,12,-5,-5,-3,-3,-2,-6,-5,-5,-4,-3,0,-2,-8,0,-2,-4,-5,-3,-8

6
Q,0,1,1,2,-5,4,2,-1,3,-2,-2,1,-1,-5,0,-1,-1,-5,-4,-2,1,3,-1,-8

7
E,0,-1,1,3,-5,2,4,0,1,-2,-3,0,-2,-5,-1,0,0,-7,-4,-2,3,3,-1,-8
```

我们发现 enumerate()函数将文件的每一行的内容提取出来，赋值给参数 line,而参数 i 对应为从 0 开始的索引。

```
    for i,line in enumerate(f):
        if i==0:
            tokens_header = line.strip().split(",")[1:]
        else:
            a1 = line[0]
            scores = line.strip().split(",")[1:]
            for j,score in enumerate(scores):
                submat[a1 + tokens_header[j]] = float(score)
```

那么首先，如果 i==0，则说明读到了文件的第一行，但文件的第一行我们前面提到是氨基酸的名字，并不是具体的数值，因此做特别的处理。这行代码是一个连续的步骤：

（1） 去掉这个字符串末尾的换行符(“\n”，看不见，但确实存在)；——>strip()

（2） 以“,”为分隔符，分隔字符串；——>split(“,”)

（3） 并提取该字符列表从索引 1 到最后的字符（即剔除开头的“#”)；——>[1:]

（4） 并将该字符列表赋值给 tokens_header 这个列表型的变量中。Tokens_header = ??

```
if i==0:
    tokens_header = line.strip().split(",")[1:]
```

读完第一行之后，for 循环继续遍历，接下来 i==1，则不再满足 if 后的条件，于是接下来执行 else 后面的语句。

读取的文件的每一行为一个字符串，作者将 line 的第一个字符取了出来，将其赋给参数 a1。

```
a1 = line[0]
```

同样的，去掉换行符，以“,”为分隔，提取列表的第 2 到最后的值，将其保存到 scores 中。

```
scores = line.strip().split(",")[1:]
```

我们的 scores 的值就是一群数值。而这一群数值，刚好是该行所对应的碱基与对应列的碱基之间替换的分值。

	A	B	C	D	E	F	G	H	I
Q	score1[0]	Score[1]	Score[3]	Score[4]	Score[5]	Score[6]	Score[7]	Score[8]	Score[9]

上图为示意图。如 Q→A,的替换的分值为 score[0]。

所以，理解了这层关系，接下来就是通过 for 循环，不断的对字典的键-值对进行构建和存储。

```
for j,score in enumerate(scores):
    submat[a1 + tokens_header[j]] = float(score)
```

这里的 a1 就是我们前面的提取的该行的第一个字符,tokes_header 就是文件首行的字符。

所以这里就相当于说 submat[“QA”] = float(score)

从整体上将，该行代码也一共使用了两个嵌套型的 for 循环：最里面的那个 for 循环读取每一行的每列，而外面的 for 循环，遍历每行。所以，两个 for 循环即可提取一个二维的表。

最后，读取文件操作的时候，记得关闭文件（很小，但非常重要的一步，有始有终）。

```
f.close()
```

最后，函数返回字典 submat。

```
return submat
```

以上就是定义 函数 load_substitution_matrix()的全部过程。

接下来就是对函数进行测试，作者在这里选择了不同的打分矩阵的文件进行构建。

（1）PAM250

```
15 s1 = "MAAQEGKKY"
16 s2 = "MSARKP"
17 submat = load_substitution_matrix("../Daten/PAM250.txt")
18 score,matrix,traceback = smith_waterman(s1,s2,submat,gap=-10)
19 print("Optimal Score: %d" % score)
20 local_traceback(s1,s2,matrix,traceback)
```

（2）PAM70

```
submat = load_substitution_matrix("../Daten/PAM70.txt")
score,matrix,traceback = smith_waterman(s1,s2,submat,gap=-10)
print("Optimal Score: %d" % score)
local_traceback(s1,s2,matrix,traceback)
```

这两个打分矩阵的区别，我们之前也介绍过了。PAM250 适用于序列变异比较大的两条序列（更能忍受序列的差异），而 PAM70 则适用于序列变异较小的序列。

从比对的结果，我们也能够发现这一点：

PAM250 的结果	PAM70 的结果
Optimal Score: 10 MAAQ MSAR	Optimal Score: 16 MAA MSA

PAM250 的比对结果中保留了 Q-R 的比对，该碱基对的比对在 PAM250 这里被认为是更能接受的（回顾：在比对的过程中四个打分值中错配这一项打分依旧较高）。

注：该测试代码中的 load_substitution_matrix()为本小题构建的函数；smith_waterman()为第五题的第一小题修改的 smith_waterman()函数，而 local_traceback()为第三题中构建的函数。

（三） 查找表函数

该函数主要的功能：给定一个字符串，以 k 个碱基（氨基酸）为单位，在序列上移动遍历，寻找在哪些位置上存在相同的 k 个碱基（氨基酸）的片段，并输出片段的初始坐标。

首先，作者同样使用了 def 关键词，定义了名为“create_loopup_table”的函数，其中输入参数有两个。其一为 seq 的字符串序列，k 为遍历字符串的过程中单位序列的长度。

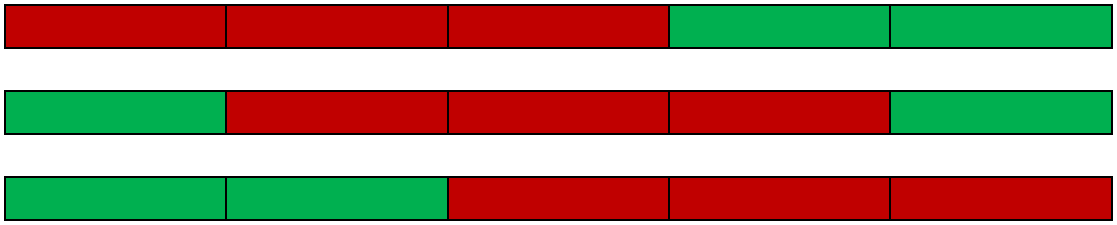
```
def create_lookup_table(seq,k=3):
```

作者在函数体中首先定义了字典 lookup。

```
    lookup = {}
```

然后就是一个 for 循环，for 循环的遍历的范围是从 0 到序列的长度-k+1 的位置。为什么是这个区间呢？其实这是一个简单的数学的问题。

假设我们要遍历的字符串的长度为 5,我们以 3 为单位进行遍历,则一共可以移动几次?



根据上述的图示，我们知道答案是 3 次，即 5-3+1=3。

即 i 的取值的范围为 0，1，2。

所以我们不难理解 range（0,len(seq)-k+1），索引从 0 开始，循环到 len(seq)-k 的位置结束。

接下来作者截取字符串的第 i 到第 i+k-1 的序列，共 k 个字符的片段存储到 tup 变量中。

```
tup = seq[i:i+k]
```

在做好上述准备工作之后，我们开始进行 if……else 的判断。

```
5           if tup in lookup:
6               tuple_list = lookup[tup]
7               tuple_list.append(i)
8               lookup[tup] = tuple_list
9               #oder in einer Zeile
10              #lookup[tup].append(i)
11          else:
12              lookup[tup] = [i]
```

如果 tup 是字典 lookup 的键，则执行 if 后面的操作；如果不是，则执行 else 语句后面的操作。

我们知道字典 lookup 在创建之初是一个空字典，因此啥也没有，所以刚开始的时候是执行的是 else 语句后面的操作。该操作实际上就是把此时表示位置的索引列表[i]（列表长度为 1，仅有一个值 i），作为值，赋给字典中名为“tup”序列的键。所以，对于第一次出现的字符片段，其执行的是 else 语句。但如果遍历字符串的过程中，发现该键在之前

就已经存到字典中了，这时就执行 if 后面的语句。该语句分为三步：

（1） 提取字典中已有的键 tup 的值，为变量 tuple_list。

（2） 将现在的索引，追加到原来的值的列表中。

（3） 将新的值重新赋值给字典中的 tup 键。

```
tuple_list = lookup[tup]
tuple_list.append(i)
lookup[tup] = tuple_list
```

最后的最后，函数返回 lookup 字典。

下面就是分别以 k=3 和 k=5 的测试结果，序列都为 s1=”ATTAAGCAATTGCA”。

```
17 create_lookup_table(s1,k=3)
31 create_lookup_table(s1,k=5)
```

我们也可以很明显的看到两者输出结果的不同。

K=3	K=5
'ATT': [0, 8], 'TTA': [1], 'TAA': [2], 'AAG': [3], 'AGC': [4], 'GCA': [5, 11],	{'ATTAA': [0], 'TTAAG': [1], 'TAAGC': [2], 'AAGCA': [3], 'AGCAA': [4], 'GCAAT': [5],

```
15 s1 = "ATTAAGCAATTGCA"
```

结果也一目了然，ATT 在 s1 序列中出现了两次，一个是索引位置为 0 时，一个是索引位置为 8 时；其余结果解释同理。

（四） 读取 fasta 文件，存放到字典中，并将字典中所有的序列计算 k=3 的查找表。

按照题目要求，以 def 为关键词，构建名为 read_fasta 的函数。函数的输入参数为 file_name，即目录下 fasta 文件的名字。

```
1  def read_fasta(filename):
```

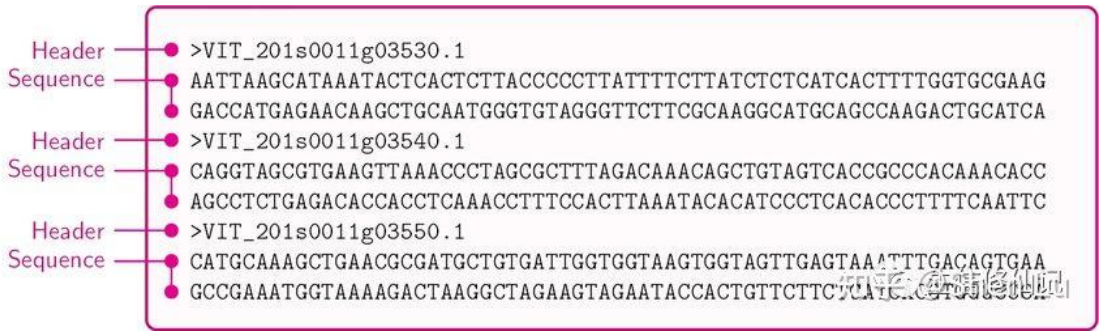
同样的，作者构建了一个空字典为 sequences。然后以可读的方式打开了 fasta 文件，并将其赋为变量 f。

```
2      sequences = {}
3      f = open(filename,'r')
```

在接下来理解下面的代码之前，我们需要明白 fasta 文件的基本的格式。

其主要由两部分组成：

（1）以“>”为开头的序列名
（2）序列信息



因此 fasta 文件是一行序列名，一行序列信息这样的格式文件。所以当我们的指针逐行读取文件时，我们也会遇到两种情况，要么读到序列，要么读到序列名的行，且序列是注释属于其前面的序列名的序列。

所以，对于 fasta 文件的每一行：

```
4      for line in f:
5          if line[0]==">":
6              sid = line[1:].strip()
7              sequences[sid] = ""
8          else:
9              sequences[sid] += line.strip()
```

如果其序列的第一个字符为“>”(为序列名行)，我们将该序列从索引 1 至最后的字符去除换行符之后，赋给变量 sid,并将 sid 为字典 sequences 的键，赋为空字符串。

接下来继续遍历文件的下一行，读取到序列信息的那一行，其第一个字符不为“>”，所

以，作者将该行序列去除换行符后，作为字典 sid 键的值存储。

注：此时 sid 的值还未重置，还为原先上一行读取时赋给的 sid 变量。因此，这样保证 fasta 文件中上一行的序列名与下一行的序列信息一一对应，以“键值对”的方式存储到字典 sequences 中。

最后，关闭文件，将字典 sequences 作为函数 read_fasta 的返回值。

测试代码的结构很简单。

使用刚刚构建的 read_fasta 函数读取 fasta 文件。得到字典 sequences。键值对分别对应序列名以及对应的序列信息。

逐个遍历字典，对于字典中的每一个键“sid”，将其对应的值（序列）作为前面构建的 create_lookup_table()函数的输入参数，以 k=3 为单位检索序列，输出结果为 table 字典。

然后就是 print 输出序列名以及对应的序列的 lookup 的结果。

```
sequences = read_fasta("uebung6.fasta")

for sid in sequences:
    table = create_lookup_table(sequences[sid],k=3)
    print("Sequence: %s" % (sid))
    print(table)
```

函数测试输出结果：

```
Sequence: Seq1
{'GAG': [0, 48, 50, 60, 81],
 'AGG': [1, 41, 57, 86],
 'GGA': [2, 13, 17, 30, 43, 47, 59, 80, 89],
 'GAT': [3, 14, 18, 44], 'ATG': [4, 15, 45],
Sequence: Seq2
{'GTT': [0], 'TTT': [1, 31], 'TTA': [2, 27], 'TAT': [3],
 'ATC': [4, 7, 12], 'TCA': [5], 'CAT': [6], 'TCC': [8, 13],
 'CCG': [9, 19], 'CGA': [10], 'GAT': [11], 'CCT': [14],
 'CTA': [15], 'TAG': [16], 'AGC': [17], 'GCC': [18],
 'CGT': [20], 'GTA': [21], 'TAC': [22, 28], 'ACG': [23],
 'CGC': [24], 'GCT': [25], 'CTT': [26, 30], 'ACT': [29]}
```

注：猜测原始的 fasta 文件，由两条序列组成。序列名为“>Sequence: Seq1”和“>Sequence: Seq2”。